User Documentation for KINSOL v2.8.2 (SUNDIALS v2.6.2)

Aaron M. Collier, Alan C. Hindmarsh, Radu Serban, and Carol S. Woodward
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

July 30, 2015

UCRL-SM-208116

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

KINSOL is part of a software family called SUNDIALS: SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [14]. This suite consists of CVODE, ARKODE, KINSOL, and IDA, and variants of these with sensitivity analysis capabilities.

KINSOL is a general-purpose nonlinear system solver based on Newton-Krylov solver technology. A fixed point iteration is also included with the release of KINSOL v.2.8.0.

1.1 Historical Background

The first nonlinear solver packages based on Newton-Krylov methods were written in FORTRAN. In particular, the NKSOL package, written at LLNL, was the first Newton-Krylov solver package written for solution of systems arising in the solution of partial differential equations [5]. This FORTRAN code made use of Newton's method to solve the discrete nonlinear systems and applied a preconditioned Krylov linear solver for solution of the Jacobian system at each nonlinear iteration. The key to the Newton-Krylov method was that the matrix-vector multiplies required by the Krylov method could effectively be approximated by a finite difference of the nonlinear system-defining function, avoiding a requirement for the formation of the actual Jacobian matrix. Significantly less memory was required for the solver as a result.

In the late 1990's, there was a push at LLNL to rewrite the nonlinear solver in C and port it to distributed memory parallel machines. Both Newton and Krylov methods are easily implemented in parallel, and this effort gave rise to the KINSOL package. KINSOL is similar to NKSOL in functionality, except that it provides for more options in the choice of linear system methods and tolerances, and has a more modular design to provide flexibility for future enhancements.

At present, KINSOL contains four Krylov methods: the GMRES (Generalized Minimal RESidual) [20], FGMRES (Flexible Generalized Minimal RESidual) [19], Bi-CGStab (Bi-Conjugate Gradient Stabilized) [21], and TFQMR (Transpose-Free Quasi-Minimal Residual) [13] linear iterative methods. As Krylov methods, these require almost no matrix storage as compared to direct methods. However, the algorithms allow for a user-supplied preconditioner matrix, and for most problems preconditioning is essential for an efficient solution. For very large nonlinear algebraic systems, the Krylov methods are preferable over direct linear solver methods, and are often the only feasible choice. Among the three Krylov methods in KINSOL, we recommend GMRES as the best overall choice. However, users are encouraged to compare all three, especially if encountering convergence failures with GMRES. Bi-CGStab and TFQMR have an advantage in storage requirements, in that the number of workspace vectors they require is fixed, while that number for GMRES depends on the desired Krylov subspace size.

For the sake of completeness in functionality, direct linear system solvers are included in KINSOL. These include methods for both dense and banded linear systems, with Jacobians that are either user-supplied or generated internally by difference quotients. KINSOL also includes interfaces to the sparse direct solvers KLU [7, 1], and the threaded sparse direct solver, SuperLU_MT [16, 9, 2].

In the process of translating NKSOL into C, the overall KINSOL organization has been changed considerably. One key feature of the KINSOL organization is that a separate module devoted to vector operations has been created. This module facilitated extension to multiprosessor environments with minimal impact on the rest of the solver. The vector module design is shared across the SUNDIALS suite. This NVECTOR module is written in terms of abstract vector operations with the actual routines attached by a particular implementation (such as serial or parallel) of NVECTOR. This abstraction allows writing the SUNDIALS solvers in a manner independent of the actual NVECTOR implementation (which can be user-supplied), as well as allowing more than one NVECTOR module linked into an executable file. SUNDIALS (and thus KINSOL) is supplied with serial, MPI-parallel, and both openMP and Pthreads thread-parallel NVECTOR implementations.

There are several motivations for choosing the C language for KINSOL. First, a general movement away from FORTRAN and toward C in scientific computing was apparent. Second, the pointer, structure, and dynamic memory allocation features in C are extremely useful in software of this complexity, with the great variety of method options offered. Finally, we prefer C over C++ for KINSOL because of the wider availability of C compilers, the potentially greater efficiency of C, and the greater ease of interfacing the solver to applications written in FORTRAN.

1.2 Changes from previous versions

Changes in v2.8.0

Two major additions were made to the globalization strategy options (`KINSol` argument `strategy`). One is fixed-point iteration, and the other is Picard iteration. Both can be accelerated by use of the Anderson acceleration method. See the relevant paragraphs in Chapter 2.

Three additions were made to the linear system solvers that are available for use with the KINSOL solver. First, in the serial case, an interface to the sparse direct solver KLU was added. Second, an interface to SuperLU_MT, the multi-threaded version of SuperLU, was added as a thread-parallel sparse direct solver option, to be used with the serial version of the NVECTOR module. As part of these additions, a sparse matrix (CSC format) structure was added to KINSOL. Finally, a variation of GMRES called Flexible GMRES was added.

Otherwise, only relatively minor modifications were made to KINSOL:

In function `KINStop`, two return values were corrected to make the values of `uu` and `fval` consistent.

A bug involving initialization of `mxnewtstep` was fixed. The error affects the case of repeated user calls to `KINSol` with no intervening call to `KINSetMaxNewtonStep`.

A bug in the increments for difference quotient Jacobian approximations was fixed in function `kinDlsBandDQJac`.

In `KINLapackBand`, the line `smu = MIN(N-1,mu+ml)` was changed to `smu = mu + ml` to correct an illegal input error for `DGBTRF/DGBTRS`.

In order to avoid possible name conflicts, the mathematical macro and function names `MIN`, `MAX`, `SQR`, `RAbs`, `RSqrt`, `RExp`, `RPowerI`, and `RPowerR` were changed to `SUNMIN`, `SUNMAX`, `SUNSQR`, `SUNRabs`, `SUNRsqrt`, `SUNRexp`, `SRpowerI`, and `SUNRpowerR`, respectively. These names occur in both the solver and in various example programs.

In the FKINSOL module, an incorrect return value `ier` in `FKINfunc` was fixed.

In the FKINSOL optional input routines `FKINSETIIN`, `FKINSETRIN`, and `FKINSETVIN`, the optional fourth argument `key_length` was removed, with hardcoded key string lengths passed to all `strncmp` tests.

In all FKINSOL examples, integer declarations were revised so that those which must match a C type `long int` are declared `INTEGER*8`, and a comment was added about the type match. All other integer declarations are just `INTEGER`. Corresponding minor corrections were made to the user guide.

Two new NVECTOR modules have been added for thread-parallel computing environments — one for openMP, denoted `NVECTOR_OPENMP`, and one for Pthreads, denoted `NVECTOR_PTHREADS`.

With this version of SUNDIALS, support and documentation of the Autotools mode of installation is being dropped, in favor of the CMake mode, which is considered more widely portable.

Changes in v2.7.0

One significant design change was made with this release: The problem size and its relatives, bandwidth parameters, related internal indices, pivot arrays, and the optional output `lsflag` have all been changed from type `int` to type `long int`, except for the problem size and bandwidths in user calls to routines specifying BLAS/LAPACK routines for the dense/band linear solvers. The function `NewIntArray` is replaced by a pair `NewIntArray`/`NewLintArray`, for `int` and `long int` arrays, respectively.

A large number of errors have been fixed. Three major logic bugs were fixed – involving updating the solution vector, updating the linesearch parameter, and a missing error return. Three minor errors were fixed – involving setting `etachoice` in the Matlab/KINSOL interface, a missing error case in `KINPrintInfo`, and avoiding an exponential overflow in the evaluation of `omega`. In each linear solver interface function, the linear solver memory is freed on an error return, and the `**Free` function now includes a line setting to NULL the main memory pointer to the linear solver memory. In the installation files, we modified the treatment of the macro SUNDIALS_USE_GENERIC_MATH, so that the parameter GENERIC_MATH_LIB is either defined (with no value) or not defined.

Changes in v2.6.0

This release introduces a new linear solver module, based on Blas and Lapack for both dense and banded matrices.

The user interface has been further refined. Some of the API changes involve: (a) a reorganization of all linear solver modules into two families (besides the already present family of scaled preconditioned iterative linear solvers, the direct solvers, including the new Lapack-based ones, were also organized into a *direct* family); (b) maintaining a single pointer to user data, optionally specified through a `Set`-type function; (c) a general streamlining of the band-block-diagonal preconditioner module distributed with the solver.

Changes in v2.5.0

The main changes in this release involve a rearrangement of the entire SUNDIALS source tree (see §3.1). At the user interface level, the main impact is in the mechanism of including SUNDIALS header files which must now include the relative path (e.g. `#include <cvode/cvode.h>`). Additional changes were made to the build system: all exported header files are now installed in separate subdirectories of the installation *include* directory.

The functions in the generic dense linear solver (`sundials_dense` and `sundials_smalldense`) were modified to work for rectangular $m \times n$ matrices ($m \leq n$), while the factorization and solution functions were renamed to `DenseGETRF`/`denGETRF` and `DenseGETRS`/`denGETRS`, respectively. The factorization and solution functions in the generic band linear solver were renamed `BandGBTRF` and `BandGBTRS`, respectively.

Changes in v2.4.0

KINSPBCG, KINSPTFQMR, KINDENSE, and KINBAND modules have been added to interface with the Scaled Preconditioned Bi-CGStab (SPBCG), Scaled Preconditioned Transpose-Free Quasi-Minimal Residual (SPTFQMR), DENSE, and BAND linear solver modules, respectively. (For details see Chapter 4.) Corresponding additions were made to the FORTRAN interface module FKINSOL. At the same time, function type names for Scaled Preconditioned Iterative Linear Solvers were added for the user-supplied Jacobian-times-vector and preconditioner setup and solve functions.

Regarding the FORTRAN interface module FKINSOL, optional inputs are now set using `FKINSETIIN` (integer inputs), `FKINSETRIN` (real inputs), and `FKINSETVIN` (vector inputs). Optional outputs are still obtained from the `IOUT` and `ROUT` arrays which are owned by the user and passed as arguments to `FKINMALLOC`.

The KINDENSE and KINBAND linear solver modules include support for nonlinear residual monitoring which can be used to control Jacobian updating.

To reduce the possibility of conflicts, the names of all header files have been changed by adding unique prefixes (`kinsol_` and `sundials_`). When using the default installation procedure, the header files are exported under various subdirectories of the target `include` directory. For more details see Appendix A.

Changes in v2.3.0

The user interface has been further refined. Several functions used for setting optional inputs were combined into a single one. Additionally, to resolve potential variable scope issues, all SUNDIALS solvers release user data right after its use. The build system has been further improved to make it more robust.

Changes in v2.2.1

The changes in this minor SUNDIALS release affect only the build system.

Changes in v2.2.0

The major changes from the previous version involve a redesign of the user interface across the entire SUNDIALS suite. We have eliminated the mechanism of providing optional inputs and extracting optional statistics from the solver through the `iopt` and `ropt` arrays. Instead, KINSOL now provides a set of routines (with prefix `KINSet`) to change the default values for various quantities controlling the solver and a set of extraction routines (with prefix `KINGet`) to extract statistics after return from the main solver routine. Similarly, each linear solver module provides its own set of `Set`- and `Get`-type routines. For more details see Chapter 4.

Additionally, the interfaces to several user-supplied routines (such as those providing Jacobian-vector products and preconditioner information) were simplified by reducing the number of arguments. The same information that was previously accessible through such arguments can now be obtained through `Get`-type functions.

Installation of KINSOL (and all of SUNDIALS) has been completely redesigned and is now based on configure scripts.

1.3 Reading this User Guide

This user guide is a combination of general usage instructions and specific examples. We expect that some readers will want to concentrate on the general instructions, while others will refer mostly to the examples, and the organization is intended to accommodate both styles.

There are different possible levels of usage of KINSOL. The most casual user, with a small nonlinear system, can get by with reading all of Chapter 2, then Chapter 4 through §4.5.3 only, and looking at examples in [6]. In a different direction, a more expert user with a nonlinear system may want to (a) use a package preconditioner (§4.7), (b) supply his/her own Jacobian or preconditioner routines (§4.6), (c) supply a new NVECTOR module (Chapter 6), or even (d) supply a different linear solver module (§3.2 and Chapter 7).

The structure of this document is as follows:

- In Chapter 2, we provide short descriptions of the numerical methods implemented by KINSOL for the solution of nonlinear systems.
- The following chapter describes the structure of the SUNDIALS suite of solvers (§3.1) and the software organization of the KINSOL solver (§3.2).
- Chapter 4 is the main usage document for KINSOL for C applications. It includes a complete description of the user interface for the solution of nonlinear algebraic systems.
- In Chapter 5, we describe FKINSOL, an interface module for the use of KINSOL with FORTRAN applications.

- Chapter 6 gives a brief overview of the generic NVECTOR module shared among the various components of SUNDIALS, and details on the four NVECTOR implementations provided with SUNDIALS: a serial implementation (§6.1), a distributed memory parallel implementation based on MPI (§6.2), and two thread-parallel implementations based on openMP (§6.3) and Pthreads (§6.4), respectively.
- Chapter 7 describes the interfaces to the linear solver modules, so that a user can provide his/her own such module.
- Chapter 8 describes in detail the generic linear solvers shared by all SUNDIALS solvers.
- Finally, in the appendices, we provide detailed instructions for the installation of KINSOL, within the structure of SUNDIALS (Appendix A), as well as a list of all the constants used for input to and output from KINSOL functions (Appendix B).

Finally, the reader should be aware of the following notational conventions in this user guide: program listings and identifiers (such as `KINInit`) within textual explanations appear in typewriter type style; fields in C structures (such as *content*) appear in italics; and packages or modules are written in all capitals. Usage and installation instructions that constitute important warnings are marked with a triangular symbol in the margin.

Acknowledgments. We wish to acknowledge the contributions to previous versions of the KINSOL code and user guide by Allan G. Taylor.

Chapter 2

Mathematical Considerations

KINSOL solves nonlinear algebraic systems in real N-space.

Using Newton's method, or the Picard iteration, one can solve

$$F(u) = 0\,, \quad F : \mathbf{R}^N \to \mathbf{R}^N\,, \tag{2.1}$$

given an initial guess u_0. Using a fixed-point iteration, the convergence of which can be improved with Anderson acceleration, one can solve

$$G(u) = u\,, \quad G : \mathbf{R}^N \to \mathbf{R}^N\,, \tag{2.2}$$

given an initial guess u_0.

Basic Newton iteration

Depending on the linear solver used, KINSOL can employ either an Inexact Newton method [4, 5, 8, 10, 15], or a Modified Newton method. At the highest level, KINSOL implements the following iteration scheme:

1. Set u_0 = an initial guess
2. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Solve $J(u_n)\delta_n = -F(u_n)$
   (b) Set $u_{n+1} = u_n + \lambda\delta_n$, $0 < \lambda \leq 1$
   (c) Test for convergence

Here, u_n is the nth iterate to u, and $J(u) = F'(u)$ is the system Jacobian. At each stage in the iteration process, a scalar multiple of the step δ_n, is added to u_n to produce a new iterate, u_{n+1}. A test for convergence is made before the iteration continues.

Newton method variants

For solving the linear system given in step 2(a), KINSOL provides several choices, including the option of a user-supplied linear solver module. The linear solver modules distributed with SUNDIALS are organized in three families, a *direct* family comprising direct linear solvers for dense or banded matrices, a *sparse* family comprising direct linear solvers for matrices stored in compressed-sparse-column format, and a *spils* family comprising scaled preconditioned iterative (Krylov) linear solvers. The methods offered through these modules are as follows:

- dense direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),

- band direct solvers, using either an internal implementation or a Blas/Lapack implementation (serial or threaded vector modules only),
- sparse direct solver interfaces, using either the KLU sparse solver library [7, 1], or the thread-enabled SuperLU_MT sparse solver library [16, 9, 2] (serial or threaded vector modules only) [Note that users will need to download and install the KLU or SuperLU_MT packages independent of KINSOL],
- SPGMR, a scaled preconditioned GMRES (Generalized Minimal Residual method) solver without restarts,
- SPFGMR, a scaled preconditioned FGMRES (Flexible Generalized Minimal Residual method) solver without restarts,
- SPBCG, a scaled preconditioned Bi-CGStab (Bi-Conjugate Gradient Stable method) solver, or
- SPTFQMR, a scaled preconditioned TFQMR (Transpose-Free Quasi-Minimal Residual method) solver.

When using one of the direct linear solvers, the linear system in 2(a) is solved exactly, thus resulting in a Modified Newton method (the Jacobian matrix is normally out of date; see below[1]). Note that the direct linear solvers (dense, band, and sparse) can only be used with the serial and threaded vector representations.

On the other hand, when using any of the iterative linear solvers (GMRES, FGMRES, Bi-CGStab, or TFQMR), the linear system in 2(a) is solved only approximately, thus resulting in an Inexact Newton method. Here right preconditioning is available by way of the preconditioning setup and solve routines supplied by the user, in which case the iterative method is applied to the linear systems $(JP^{-1})(P\delta) = -F$, where P denotes the right preconditioning matrix.

Jacobian information update strategy

In general, unless specified otherwise by the user, KINSOL strives to update Jacobian information (the actual system Jacobian J in the case of direct linear solvers, or the preconditioner matrix P in the case of iterative linear solvers) as infrequently as possible to balance the high costs of matrix operations against other costs. Specifically, these updates occur when:

- the problem is initialized,
- $\|\lambda\delta_{n-1}\|_{D_u,\infty} > 1.5$ (Inexact Newton only),
- `mbset`= 10 nonlinear iterations have passed since the last update,
- the linear solver failed recoverably with outdated Jacobian information,
- the global strategy failed with outdated Jacobian information, or
- $\|\lambda\delta_n\|_{D_u,\infty} <$ STEPTOL with outdated Jacobian information.

KINSOL allows, through optional solver inputs, changes to the above strategy. Indeed, the user can disable the initial Jacobian information evaluation or change the default value of `mbset`, the number of nonlinear iterations after which a Jacobian information update is enforced.

[1] KINSOL allows the user to enforce a Jacobian evaluation at each iteration thus allowing for an Exact Newton iteration.

Scaling

To address the case of ill-conditioned nonlinear systems, KINSOL allows prescribing scaling factors both for the solution vector and for the residual vector. For scaling to be used, the user should supply values D_u, which are diagonal elements of the scaling matrix such that $D_u u_n$ has all components roughly the same magnitude when u_n is close to a solution, and D_F, which are diagonal scaling matrix elements such that $D_F F$ has all components roughly the same magnitude when u_n is not too close to a solution. In the text below, we use the following scaled norms:

$$\|z\|_{D_u} = \|D_u z\|_2, \quad \|z\|_{D_F} = \|D_F z\|_2, \quad \|z\|_{D_u,\infty} = \|D_u z\|_\infty, \quad \text{and} \quad \|z\|_{D_F,\infty} = \|D_F z\|_\infty \tag{2.3}$$

where $\|\cdot\|_\infty$ is the max norm. When scaling values are provided for the solution vector, these values are automatically incorporated into the calculation of the perturbations used for the default difference quotient approximations for Jacobian information; see (2.7) and (2.9) below.

Globalization strategy

Two methods of applying a computed step δ_n to the previously computed solution vector are implemented. The first and simplest is the standard Newton strategy which applies step 2(b) as above with λ always set to 1. The other method is a global strategy, which attempts to use the direction implied by δ_n in the most efficient way for furthering convergence of the nonlinear problem. This technique is implemented in the second strategy, called Linesearch. This option employs both the α and β conditions of the Goldstein-Armijo linesearch given in [10] for step 2(b), where λ is chosen to guarantee a sufficient decrease in F relative to the step length as well as a minimum step length relative to the initial rate of decrease of F. One property of the algorithm is that the full Newton step tends to be taken close to the solution.

KINSOL implements a backtracking algorithm to first find the value λ such that $u_n + \lambda\delta_n$ satisfies the sufficient decrease condition (or α-condition)

$$F(u_n + \lambda\delta_n) \leq F(u_n) + \alpha \nabla F(u_n)\lambda\delta_n ,$$

where $\alpha = 10^{-4}$. Although backtracking in itself guarantees that the step is not too small, KINSOL secondly relaxes λ to satisfy the so-called β-condition (equivalent to Wolfe's curvature condition):

$$F(u_n + \lambda\delta_n) \geq F(u_n) + \beta \nabla F(u_n)\lambda\delta_n ,$$

where $\beta = 0.9$. During this second phase, λ is allowed to vary in the interval $[\lambda_{min}, \lambda_{max}]$ where

$$\lambda_{min} = \frac{\text{STEPTOL}}{\|\bar{\delta}_n\|_\infty} , \quad \bar{\delta}_n^j = \frac{\delta_n^j}{1/D_u^j + |u^j|} ,$$

and λ_{max} corresponds to the maximum feasible step size at the current iteration (typically $\lambda_{max} = \text{STEPMAX}/\|\delta_n\|_{D_u}$). In the above expressions, v^j denotes the jth component of a vector v.

For more details, the reader is referred to [10].

Nonlinear iteration stopping criteria

Stopping criteria for the Newton method are applied to both of the nonlinear residual and the step length. For the former, the Newton iteration must pass a stopping test

$$\|F(u_n)\|_{D_F,\infty} < \text{FTOL} ,$$

where FTOL is an input scalar tolerance with a default value of $U^{1/3}$. Here U is the machine unit roundoff. For the latter, the Newton method will terminate when the maximum scaled step is below a given tolerance

$$\|\lambda\delta_n\|_{D_u,\infty} < \text{STEPTOL} ,$$

where STEPTOL is an input scalar tolerance with a default value of $U^{2/3}$. Only the first condition (small residual) is considered a successful completion of KINSOL. The second condition (small step) may indicate that the iteration is stalled near a point for which the residual is still unacceptable.

Additional constraints

As a user option, KINSOL permits the application of inequality constraints, $u^i > 0$ and $u^i < 0$, as well as $u^i \geq 0$ and $u^i \leq 0$, where u^i is the ith component of u. Any such constraint, or no constraint, may be imposed on each component. KINSOL will reduce step lengths in order to ensure that no constraint is violated. Specifically, if a new Newton iterate will violate a constraint, the maximum step length along the Newton direction that will satisfy all constraints is found, and δ_n in Step 2(b) is scaled to take a step of that length.

Residual monitoring for Modified Newton method

When using a Modified Newton method (i.e. when a direct linear solver is used), in addition to the strategy described above for the update of the Jacobian matrix, KINSOL also provides an optional nonlinear residual monitoring scheme to control when the system Jacobian is updated. Specifically, a Jacobian update will also occur when `mbsetsub`$= 5$ nonlinear iterations have passed since the last update and

$$\|F(u_n)\|_{D_F} > \omega \|F(u_m)\|_{D_F} \, ,$$

where u_n is the current iterate and u_m is the iterate at the last Jacobian update. The scalar ω is given by

$$\omega = \min \left(\omega_{min} \, e^{\max(0,\rho-1)}, \omega_{max} \right) \, , \tag{2.4}$$

with ρ defined as

$$\rho = \frac{\|F(u_n)\|_{D_F}}{\text{FTOL}} \, , \tag{2.5}$$

where FTOL is the input scalar tolerance discussed before. Optionally, a constant value ω_{const} can be used for the parameter ω.

The constants controlling the nonlinear residual monitoring algorithm can be changed from their default values through optional inputs to KINSOL. These include the parameters ω_{min} and ω_{max}, the constant value ω_{const}, and the threshold `mbsetsub`.

Stopping criteria for iterative linear solvers

When using an Inexact Newton method (i.e. when an iterative linear solver is used), the convergence of the overall nonlinear solver is intimately coupled with the accuracy with which the linear solver in 2(a) above is solved. KINSOL provides three options for stopping criteria for the linear system solver, including the two algorithms of Eisenstat and Walker [11]. More precisely, the Krylov iteration must pass a stopping test

$$\|J\delta_n + F\|_{D_F} < (\eta_n + U)\|F\|_{D_F} \, ,$$

where η_n is one of:

Eisenstat and Walker Choice 1

$$\eta_n = \frac{\left| \, \|F(u_n)\|_{D_F} - \|F(u_{n-1}) + J(u_{n-1})\delta_n\|_{D_F} \, \right|}{\|F(u_{n-1})\|_{D_F}} \, ,$$

Eisenstat and Walker Choice 2

$$\eta_n = \gamma \left(\frac{\|F(u_n)\|_{D_F}}{\|F(u_{n-1})\|_{D_F}} \right)^{\alpha} \, ,$$

where default values of γ and α are 0.9 and 2, respectively.

Constant η

$$\eta_n = constant,$$

with 0.1 as the default.

The default strategy is "Eisenstat and Walker Choice 1". For both options 1 and 2, appropriate safeguards are incorporated to ensure that η does not decrease too quickly [11].

Difference quotient Jacobian approximations

With the direct dense and band methods, the Jacobian may be supplied by a user routine, or approximated by difference quotients, at the user's option. In the latter case, we use the usual approximation

$$J^{ij} = [F^i(u + \sigma_j e^j) - F^i(u)]/\sigma_j \,. \tag{2.6}$$

The increments σ_j are given by

$$\sigma_j = \sqrt{U} \;\max\left\{|u^j|, 1/D_u^j\right\} \,. \tag{2.7}$$

In the dense case, this scheme requires N evaluations of F, one for each column of J. In the band case, the columns of J are computed in groups, by the Curtis-Powell-Reid algorithm, with the number of F evaluations equal to the bandwidth. The parameter U above can (optionally) be replaced by a user-specified value, `relfunc`.

We note that with the sparse direct solvers, the Jacobian *must* be supplied by a user routine in compressed-sparse-column format, i.e. it is not approximated internally within KINSOL.

In the case of a Krylov method, Jacobian information is needed only as matrix-vector products Jv. If a routine for Jv is not supplied, these products are approximated by directional difference quotients as

$$J(u)v \approx [F(u + \sigma v) - F(u)]/\sigma \,, \tag{2.8}$$

where u is the current approximation to a root of (2.1), and σ is a scalar. The choice of σ is taken from [5] and is given by

$$\sigma = \frac{\max\{|u^T v|, u_{typ}^T |v|\}}{\|v\|_2^2} \mathrm{sign}(u^T v)\sqrt{U} \,, \tag{2.9}$$

where u_{typ} is a vector of typical values for the absolute values of the solution (and can be taken to be inverses of the scale factors given for u as described below). This formula is suitable for *scaled* vectors u and v, and so is applied to $D_u u$ and $D_u v$. The parameter U above can (optionally) be replaced by a user-specified value, `relfunc`. Convergence of the Newton method is maintained as long as the value of σ remains appropriately small, as shown in [4].

Basic Fixed Point iteration

The basic fixed-point iteration scheme implemented in KINSOL is given by:

1. Set $u_0 =$ an initial guess
2. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Set $u_{n+1} = G(u_n)$.
   (b) Test for convergence.

Here, u_n is the nth iterate to u. At each stage in the iteration process, function G is applied to the current iterate to produce a new iterate, u_{n+1}. A test for convergence is made before the iteration continues.

For Picard iteration, as implemented in KINSOL, we consider a special form of the nonlinear function F, such that $F(u) = Lu - N(u)$, where L is a constant nonsingular matrix and N is (in general) nonlinear. Then the fixed-point function G is defined as $G(u) = u - L^{-1}F(u)$. The Picard iteration is given by:

1. Set $u_0 =$ an initial guess
2. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Set $u_{n+1} = G(u_n) = u_n - L^{-1}F(u_n)$.
   (b) Test $F(u_{n+1})$ for convergence.

Here, u_n is the nth iterate to u. Within each iteration, the Picard step is computed then added to u_n to produce the new iterate. Next, the nonlinear residual function is evaluated at the new iterate, and convergence is checked. Noting that $L^{-1}N(u) = u - L^{-1}F(u)$, the above iteration can be written in the same form as a Newton iteration except that here, L is in the role of the Jacobian. Within KINSOL, however, we leave this in a fixed-point form as above. For more information, see p. 182 of [18].

Anderson Acceleration

The Picard and fixed point methods can be significantly accelerated using Anderson's method [3, 22, 12, 17]. Anderson acceleration can be formulated as follows:

1. Set $u_0 =$ an initial guess and $m \geq 1$
2. Set $u_1 = G(u_0)$
3. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Set $m_n = \min\{m, n\}$
   (b) Set $F_n = (f_{n-m_n}, \ldots, f_n)$, where $f_i = G(u_i) - u_i$
   (c) Determine $\alpha^{(n)} = (\alpha_0^{(n)}, \ldots, \alpha_{m_n}^{(n)})$ that solves $\min_\alpha \|F_n \alpha^T\|_2$ such that $\sum_{i=0}^{m_n} \alpha_i = 1$
   (d) Set $u_{n+1} = \sum_{i=0}^{m_n} \alpha_i^{(n)} G(u_{n-m_n+i})$
   (e) Test for convergence

It has been implemented in KINSOL by turning the constrained linear least-squares problem in Step (c) into an unconstrained one leading to the algorithm given below:

1. Set $u_0 =$ an initial guess and $m \geq 1$
2. Set $u_1 = G(u_0)$
3. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Set $m_n = \min\{m, n\}$
   (b) Set $\Delta F_n = (\Delta f_{n-m_n}, \ldots, \Delta f_{n-1})$, where $\Delta f_i = f_{i+1} - f_i$ and $f_i = G(u_i) - u_i$
   (c) Determine $\gamma^{(n)} = (\gamma_0^{(n)}, \ldots, \gamma_{m_n-1}^{(n)})$ that solves $\min_\gamma \|f_n - \Delta F_n \gamma^T\|_2$
   (d) Set $u_{n+1} = G(u_n) - \sum_{i=0}^{m_n-1} \gamma_i^{(n)} \Delta g_{n-m_n+i}$ with $\Delta g_i = G(u_{i+1}) - G(u_i)$
   (e) Test for convergence

The least-squares problem in (c) is solved by applying a QR factorization to $\Delta F_n = Q_n R_n$ and solving $R_n \gamma = Q_n^T f_n$.

Fixed-point - Anderson Acceleration Stopping Criterion

The default stopping criterion is

$$\|G(u_{n+1}) - u_{n+1}\|_{D_F,\infty} < \text{GTOL}\,,$$

where D_F is a user-defined diagonal matrix that can be the identity or a scaling matrix chosen so that the components of $D_F(G(u) - u)$ have roughly the same order of magnitude. Note that when using Anderson acceleration, convergence is checked after the acceleration is applied.

Picard - Anderson Acceleration Stopping Criterion

The default stopping criterion is

$$\|F(u_{n+1})\|_{D_F,\infty} < \text{FTOL}\,,$$

where D_F is a user-defined diagonal matrix that can be the identity or a scaling matrix chosen so that the components of $D_F F(u)$ have roughly the same order of magnitude. Note that when using Anderson acceleration, convergence is checked after the acceleration is applied.

Chapter 3

Code Organization

3.1 SUNDIALS organization

The family of solvers referred to as SUNDIALS consists of the solvers CVODE and ARKODE (for ODE systems), KINSOL (for nonlinear algebraic systems), and IDA (for differential-algebraic systems). In addition, SUNDIALS also includes variants of CVODE and IDA with sensitivity analysis capabilities (using either forward or adjoint methods), called CVODES and IDAS, respectively.

The various solvers of this family share many subordinate modules. For this reason, it is organized as a family, with a directory structure that exploits that sharing (see Fig. 3.1). The following is a list of the solver packages presently available, and the basic functionality of each:

- CVODE, a solver for stiff and nonstiff ODE systems $dy/dt = f(t, y)$ based on Adams and BDF methods;
- CVODES, a solver for stiff and nonstiff ODE systems with sensitivity analysis capabilities;
- ARKODE, a solver for ODE systems $Mdy/dt = f(t, y)$ based on additive Runge-Kutta methods;
- IDA, a solver for differential-algebraic systems $F(t, y, \dot{y}) = 0$ based on BDF methods;
- IDAS, a solver for differential-algebraic systems with sensitivity analysis capabilities;
- KINSOL, a solver for nonlinear algebraic systems $F(u) = 0$.

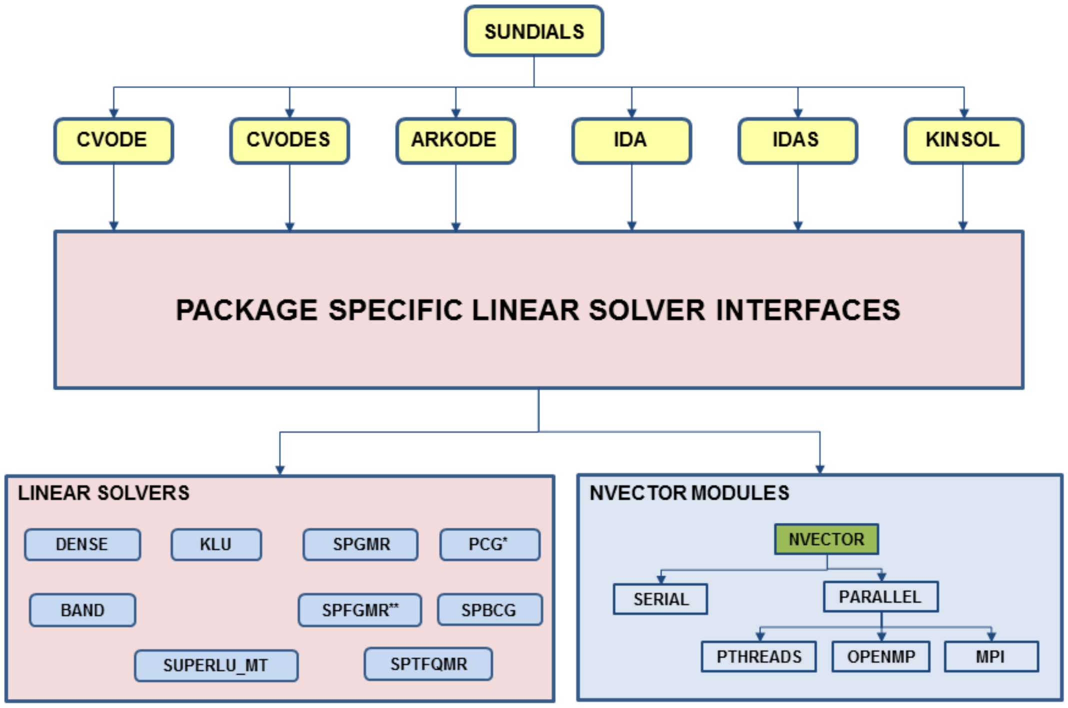
3.2 KINSOL organization

The KINSOL package is written in the ANSI C language. This section summarizes the basic structure of the package, although knowledge of this structure is not necessary for its use.

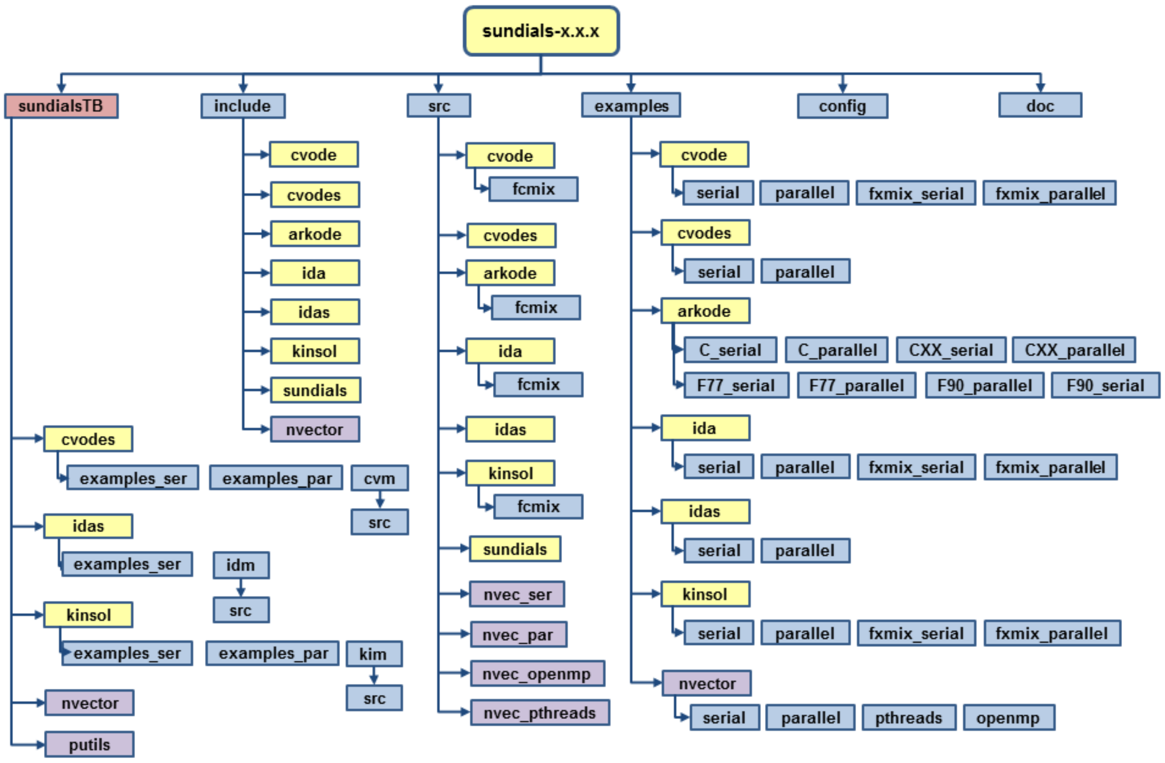
The overall organization of the KINSOL package is shown in Figure 3.2. The central solver module, implemented in the files `kinsol.h`, `kinsol_impl.h` and `kinsol.c`, deals with the solution of a nonlinear algebraic system using either an Inexact Newton method or a line search method for the global strategy. Although this module contains logic for the Newton iteration, it has no knowledge of the method used to solve the linear systems that arise. For any given user problem, one of the linear system modules is specified, and is then invoked as needed.

At present, the package includes the following seven KINSOL linear algebra modules, organized into two families. The *direct* family of linear solvers provides solvers for the direct solution of linear systems with dense, banded, or sparse matrices and includes:

- KINDENSE: LU factorization and backsolving with dense matrices (using either an internal implementation or Blas/Lapack);
- KINBAND: LU factorization and backsolving with banded matrices (using either an internal implementation or Blas/Lapack);



(a) High-level diagram (note that none of the Lapack-based linear solver modules are represented.)
* only applies to ARKODE
** only applies to ARKODE and KINSOL



(b) Directory structure of the source tree

Figure 3.1: Organization of the SUNDIALS suite

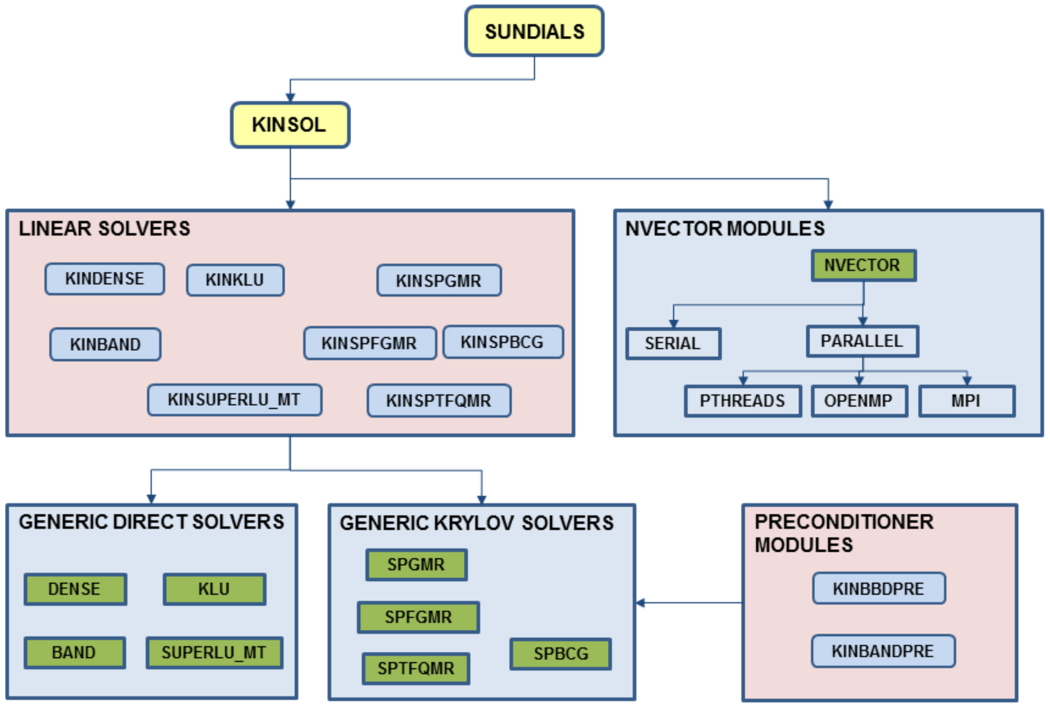


Figure 3.2: Overall structure diagram of the KINSOL package. Modules specific to KINSOL are distinguished by rounded boxes, while generic solver and auxiliary modules are in rectangular boxes. Grayed boxes refer to the encompassing SUNDIALS structure. Note that the direct linear solvers using Lapack implementations are not explicitly represented. Note also that the KLU and SuperLU_MT support is through interfaces to packages. Users will need to download and compile those packages independently.

- KINKLU: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the KLU linear solver library [7, 1] (KLU to be downloaded and compiled by user independent of KINSOL);
- KINSUPERLUMT: LU factorization and backsolving with compressed-sparse-column (CSC) matrices using the threaded SuperLU_MT linear solver library [16, 9, 2] (SuperLU_MT to be downloaded and compiled by user independent of KINSOL).

The *spils* family of linear solvers providess scaled preconditioned iterative linear solvers and includes:

- KINSPGMR: scaled preconditioned GMRES method;
- KINSPBCG: scaled preconditioned Bi-CGStab method;
- KINSPTFQMR: scaled preconditioned TFQMR method.

The set of linear solver modules distributed with KINSOL is intended to be expanded in the future as new algorithms are developed. Note that users wishing to employ KLU or SuperLU_MT will need to download and install these libraries independent of SUNDIALS. SUNDIALS provides only the interfaces between itself and these libraries.

In the case of the direct methods KINDENSE and KINBAND the package includes an algorithm for the approximation of the Jacobian by difference quotients, but the user also has the option of supplying the Jacobian (or an approximation to it) directly. When using the sparse direct linear solvers KINKLU and KINSUPERLUMT, the user must supply a routine for the Jacobian (or an approximation to it) in

CSC format, since standard difference quotient approximations do not leverage the inherent sparsity of the problem. In the case of the Krylov methods KINSPGMR, KINSPBCG and KINSPTFQMR, the package includes an algorithm for the approximation by difference quotients of the product between the Jacobian matrix and a vector of appropriate length. Again, the user has the option of providing a routine for this operation. For the Krylov methods, the preconditioning must be supplied by the user, in two phases: setup (preprocessing of Jacobian data) and solve.

Each KINSOL linear solver module consists of four routines, devoted to (1) memory allocation and initialization, (2) setup of the matrix data involved, (3) solution of the system, and (4) freeing of memory. The setup and solution phases are separate because the evaluation of Jacobians and preconditioners is done only periodically during the solution, as required to achieve convergence. The call list within the central KINSOL module to each of the associated functions is fixed, thus allowing the central module to be completely independent of the linear system method.

These modules are also decomposed in another way. Each of the linear solver modules (KINDENSE, etc.) consists of an interface built on top of a generic linear system solver (DENSE etc.). The interface deals with the use of the particular method in the KINSOL context, whereas the generic solver is independent of the context. While some of the generic linear system solvers (DENSE, BAND, SPGMR, SPFGMR, SPBCG, and SPTFQMR) were written with SUNDIALS in mind, they are intended to be usable anywhere as general-purpose solvers. This separation also allows for any generic solver to be replaced by an improved version, with no necessity to revise the KINSOL package elsewhere.

KINSOL also provides a preconditioner module called KINBBDPRE for use with any of the Krylov iterative liear solvers. It works in conjunction with NVECTOR_PARALLEL and generates a preconditioner that is a block-diagonal matrix with each block being a band matrix, as further described in §4.7.

All state information used by KINSOL to solve a given problem is saved in a structure, and a pointer to that structure is returned to the user. There is no global data in the KINSOL package, and so, in this respect, it is reentrant. State information specific to the linear solver is saved in a separate structure, a pointer to which resides in the KINSOL memory structure. The reentrancy of KINSOL was motivated by the anticipated multi-computer extension.

Chapter 4

Using KINSOL for C Applications

This chapter is concerned with the use of KINSOL for the solution of nonlinear systems. The following subsections treat the header files, the layout of the user's main program, description of the KINSOL user-callable routines, and user-supplied functions. The sample programs described in the companion document [6] may also be helpful. Those codes may be used as templates (with the removal of some lines involved in testing), and are included in the KINSOL package.

Users with applications written in FORTRAN77 should see Chapter 5, which describes the FORTRAN/C interface module.

The user should be aware that not all linear solver modules are compatible with all NVECTOR implementations. For example, NVECTOR_PARALLEL is not compatible with the direct dense, direct band or direct sparse linear solvers since these linear solver modules need to form the complete system Jacobian. The following KINSOL modules can only be used with NVECTOR_SERIAL, NVECTOR_OPENMP or NVECTOR_PTHREADS: KINDENSE, KINBAND, KINKLU and KINSUPERLUMT. It is not recommended to use a threaded vector module with SuperLU_MT unless it is the NVECTOR_OPENMP module and SuperLU_MT is also compiled with openMP. The preconditioner module KINBBDPRE can only be used with NVECTOR_PARALLEL.

KINSOL uses various constants for both input and output. These are defined as needed in this chapter, but for convenience are also listed separately in Appendix B.

4.1 Access to library and header files

At this point, it is assumed that the installation of KINSOL, following the procedure described in Appendix A, has been completed successfully.

Regardless of where the user's application program resides, its associated compilation and load commands must make reference to the appropriate locations for the library and header files required by KINSOL. The relevant library files are

- *libdir*/`libsundials_kinsol`.*lib*,
- *libdir*/`libsundials_nvec*`.*lib* (one to four files),

where the file extension .*lib* is typically `.so` for shared libraries and `.a` for static libraries. The relevant header files are located in the subdirectories

- *incdir*/`include`
- *incdir*/`include/kinsol`
- *incdir*/`include/sundials`

The directories *libdir* and *incdir* are the install library and include directories, respectively. For a default installation, these are *builddir*/`lib` and *builddir*/`include`, respectively, where *builddir* was defined in Appendix A.

4.2 Data types

The `sundials_types.h` file contains the definition of the type `realtype`, which is used by the SUNDIALS solvers for all floating-point data. The type `realtype` can be `float`, `double`, or `long double`, with the default being `double`. The user can change the precision of the SUNDIALS solvers arithmetic at the configuration stage (see §A.1.2).

Additionally, based on the current precision, `sundials_types.h` defines `BIG_REAL` to be the largest value representable as a `realtype`, `SMALL_REAL` to be the smallest value representable as a `realtype`, and `UNIT_ROUNDOFF` to be the difference between 1.0 and the minimum `realtype` greater than 1.0.

Within SUNDIALS, real constants are set by way of a macro called `RCONST`. It is this macro that needs the ability to branch on the definition `realtype`. In ANSI C, a floating-point constant with no suffix is stored as a `double`. Placing the suffix "F" at the end of a floating point constant makes it a `float`, whereas using the suffix "L" makes it a `long double`. For example,

```
#define A 1.0
#define B 1.0F
#define C 1.0L
```

defines `A` to be a `double` constant equal to 1.0, `B` to be a `float` constant equal to 1.0, and `C` to be a `long double` constant equal to 1.0. The macro call `RCONST(1.0)` automatically expands to `1.0` if `realtype` is `double`, to `1.0F` if `realtype` is `float`, or to `1.0L` if `realtype` is `long double`. SUNDIALS uses the `RCONST` macro internally to declare all of its floating-point constants.

A user program which uses the type `realtype` and the `RCONST` macro to handle floating-point constants is precision-independent except for any calls to precision-specific standard math library functions. (Our example programs use both `realtype` and `RCONST`.) Users can, however, use the type `double`, `float`, or `long double` in their code (assuming that this usage is consistent with the typedef for `realtype`). Thus, a previously existing piece of ANSI C code can use SUNDIALS without modifying the code to use `realtype`, so long as the SUNDIALS libraries use the correct precision (for details see §A.1.2).

4.3 Header files

The calling program must include several header files so that various macros and data types can be used. The header file that is always required is:

- `kinsol.h`, the header file for KINSOL, which defines several types and various constants, and includes function prototypes.

`kinsol.h` also includes `sundials_types.h`, which defines the types `realtype` and `booleantype` and constants `FALSE` and `TRUE`.

The calling program must also include an NVECTOR implementation header file (see Chapter 6 for details). For the two NVECTOR implementations that are included in the KINSOL package, the corresponding header files are:

- `nvector_serial.h`, which defines the serial implementation, NVECTOR_SERIAL;
- `nvector_parallel.h`, which defines the parallel MPI implementation, NVECTOR_PARALLEL,
- `nvector_openmp.h`, which defines the shared memory parallel openMP implementation,
- `nvector_pthreads.h`, which defines the shared memory parallel Pthreads implementation.

Note that these files include in turn the header file `sundials_nvector.h`, which defines the abstract `N_Vector` type.

Finally, a linear solver module header file is required. The header files corresponding to the various linear solver options in KINSOL are:

- `kinsol_dense.h`, which is used with the dense direct linear solver;

- `kinsol_band.h`, which is used with the band direct linear solver;
- `kinsol_lapack.h`, which is used with Lapack implementations of dense or band direct linear solvers;
- `kinsol_klu.h`, which is used with the KLU sparse direct linear solver;
- `kinsol_superlumt.h`, which is used with the SuperLU_MT threaded sparse direct linear solver;
- `kinsol_spgmr.h`, which is used with the Krylov solver SPGMR;
- `kinsol_spfgmr.h`, which is used with the Krylov solver SPFGMR;
- `kinsol_spbcgs.h`, which is used with the Krylov solver SPBCG;
- `kinsol_sptfqmr.h`, which is used with the Krylov solver SPTFQMR;

The header files for the dense and banded linear solvers (both internal and Lapack) include the file `kinsol_direct.h` which defines common functions. This in turn includes a file (`sundials_direct.h`) which defines the matrix type for these direct linear solvers (`DlsMat`), as well as various functions and macros acting on such matrices.

The header files for the KLU and SuperLU_MT sparse linear solvers include the file `kinsol_sparse.h`, which defines common functions. This in turn includes a file (`sundials_sparse.h`) which defines the matrix type for these sparse direct linear solvers (`SlsMat`), as well as various functions and macros acting on such matrices.

The header files for the Krylov iterative solvers include `kinsol_spils.h` which defined common functions and which in turn includes a header file (`sundials_iterative.h`) which enumerates the kind of preconditioning and for the choices for the Gram-Schmidt process for SPGMR.

Other headers may be needed, according to the choice of preconditioner, etc. For example, in the `kinFoodWeb_kry_p` example (see [6]), preconditioning is done with a block-diagonal matrix. For this, even though the KINSPGMR linear solver is used, the header `sundials_dense.h` is included for access to the underlying generic dense linear solver.

4.4 A skeleton of the user's main program

The following is a skeleton of the user's main program (or calling program) for the solution of a nonlinear problem. Some steps are independent of the NVECTOR implementation used; where this is not the case, usage specifications are given for the implementations provided with KINSOL: Steps marked [**P**] correspond to NVECTOR_PARALLEL, steps marked [**O**] correspond to NVECTOR_OPENMP, steps marked [**T**] correspond to NVECTOR_PTHREADS, while steps marked [**S**] correspond to NVECTOR_SERIAL.

1. **[P] Initialize MPI**

   Call `MPI_Init(&argc, &argv)` to initialize MPI if used by the user's program, aside from the internal use in NVECTOR_PARALLEL. Here `argc` and `argv` are the command line argument counter and array received by `main`.

2. **Set problem dimensions**

   [**S**], [**O**], [**T**] Set `N`, the problem size N.

   [**O**], [**T**] Set `num_threads`, the number of threads to use within the threaded vector functions.

   [**P**] Set `Nlocal`, the local vector length (the sub-vector length for this process). Set `N`, the global vector length (the problem size N, and the sum of all the values of `Nlocal`). Set the active set of processes.

   Note: The variables `N` and `Nlocal` should be of type `long int`. The variable `num_threads` should be of type `int`.

3. **Set vector with initial guess**

   To set the vector `u` of initial values, use functions defined by a particular NVECTOR implementation. If a `realtype` array `udata` already exists, containing the initial guess of u_0, make the call:

   [**S**] `u = N_VMake_Serial(N, udata);`

   [**O**] `y0 = N_VMake_OpenMP(N, num_threads, ydata);`

   [**T**] `y0 = N_VMake_Pthreads(N, num_threads, ydata);`

   [**P**] `u = N_VMake_Parallel(comm, Nlocal, N, udata);`

   Otherwise, make the call:

   [**S**] `u = N_VNew_Serial(N);`

   [**O**] `y0 = N_VNew_OpenMP(N, num_threads);`

   [**T**] `y0 = N_VNew_Pthreads(N, num_threads);`

   [**P**] `u = N_VNew_Parallel(comm, Nlocal, N);`

   and load initial values into the structure defined by:

   [**S**] `NV_DATA_S(u)`

   [**O**] `NV_DATA_OMP(y0)`

   [**T**] `NV_DATA_PT(y0)`

   [**P**] `NV_DATA_P(u)`

   Here `comm` is the MPI communicator, set in one of two ways: If a proper subset of active processes is to be used, `comm` must be set by suitable MPI calls. Otherwise, to specify that all processes are to be used, `comm` must be `MPI_COMM_WORLD`.

4. **Create KINSOL object**

   Call `kin_mem = KINCreate()` to create the KINSOL memory block. `KINCreate` returns a pointer to the KINSOL memory structure. See §4.5.1 for details.

5. **Set optional inputs**

   Call `KINSet*` routines to change from their default values any optional inputs that control the behavior of KINSOL. See §4.5.4 for details.

6. **Allocate internal memory**

   Call `KINInit(...)` to specify the problem defining function F, allocate internal memory for KINSOL, and initialize KINSOL. `KINInit` returns a flag to indicate success or an illegal argument value. See §4.5.1 for details.

7. **Attach linear solver module**

   Initialize the linear solver module with one of the following calls (for details see §4.5.2).

   [**S**], [**O**], [**T**] `ier = KINDense(...);`

   [**S**], [**O**], [**T**] `ier = KINBand(...);`

   [**S**], [**O**], [**T**] `ier = KINLapackDense(...);`

   [**S**], [**O**], [**T**] `ier = KINLapackBand(...);`

   [**S**], [**O**], [**T**] `ier = KINKLU(...);`

   [**S**], [**O**], [**T**] `ier = KINSuperLUMT(...);`

   `ier = KINSpgmr(...);`

   `ier = KINSpfgmr(...);`

`ier = KINSpbcg(...);`

`ier = KINSptfqmr(...);`

8. **Set linear solver optional inputs**

   Call `KIN*Set*` functions from the selected linear solver module to change optional inputs specific to that linear solver. See §4.5.4 for details.

9. **Solve problem**

   Call `ier = KINSol(...)` to solve the nonlinear problem for a given initial guess. See §4.5.3 for details.

10. **Get optional outputs**

    Call `KINGet*` and `KIN*Get*` functions to obtain optional output. See §4.5.5 for details.

11. **Deallocate memory for solution vector**

    Upon completion of the solution, deallocate memory for the vector `u` by calling the destructor function defined by the NVECTOR implementation:

    [**S**] `N_VDestroy_Serial(u);`

    [**O**] `N_VDestroy_OpenMP(y);`

    [**T**] `N_VDestroy_Pthreads(y);`

    [**P**] `N_VDestroy_Parallel(u);`

12. **Free solver memory**

    Call `KINFree(&kin_mem)` to free the memory allocated for KINSOL.

13. **[P] Finalize MPI**

    Call `MPI_Finalize()` to terminate MPI.

4.5 User-callable functions

This section describes the KINSOL functions that are called by the user to set up and solve a nonlinear problem. Some of these are required. However, starting with §4.5.4, the functions listed involve optional inputs/outputs or restarting, and those paragraphs can be skipped for a casual use of KINSOL. In any case, refer to §4.4 for the correct order of these calls.

The return flag (when present) for each of these routines is a negative integer if an error occurred, and non-negative otherwise.

4.5.1 KINSOL initialization and deallocation functions

The following three functions must be called in the order listed. The last one is to be called only after the problem solution is complete, as it frees the KINSOL memory block created and allocated by the first two calls.

`KINCreate`

Call	`kin_mem = KINCreate();`
Description	The function `KINCreate` instantiates a KINSOL solver object.
Arguments	This function has no arguments.
Return value	If successful, `KINCreate` returns a pointer to the newly created KINSOL memory block (of type `void *`). If an error occurred, `KINCreate` prints an error message to `stderr` and returns `NULL`.

`KINInit`

Call	`flag = KINInit(kin_mem, func, tmpl);`
Description	The function `KINInit` specifies the problem-defining function, allocates internal memory, and initializes KINSOL.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block returned by `KINCreate`. `func` (`KINSysFn`) is the C function which computes the system function F (or $G(u)$ for fixed-point iteration) in the nonlinear problem. This function has the form `func(u, fval, user_data)`. (For full details see §4.6.1.) `tmpl` (`N_Vector`) is any `N_Vector` (e.g. the initial guess vector `u`) which is used as a template to create (by cloning) necessary vectors in `kin_mem`.
Return value	The return value `flag` (of type `int`) will be one of the following: `KIN_SUCCESS` The call to `KINInit` was successful. `KIN_MEM_NULL` The KINSOL memory block was not initialized through a previous call to `KINCreate`. `KIN_MEM_FAIL` A memory allocation request has failed. `KIN_ILL_INPUT` An input argument to `KINInit` has an illegal value.
Notes	If an error occurred, `KINInit` sends an error message to the error handler function.

`KINFree`

Call	`KINFree(&kin_mem);`
Description	The function `KINFree` frees the memory allocated by a previous call to `KINCreate`.
Arguments	The argument is the address of the pointer to the KINSOL memory block returned by `KINCreate` (of type `void *`).
Return value	The function `KINFree` has no return value.

4.5.2 Linear solver specification functions

As previously explained, Newton and Picard iterations require the solution of linear systems of the form $J\delta = -F$. There are several KINSOL linear solvers currently available for this task: KINDENSE, KINBAND, KINKLU, KINSUPERLUMT, KINSPGMR, KINSPFGMR, KINSPBCG, and KINSPTFQMR.

The first two linear solvers are direct and derive their names from the type of approximation used for the Jacobian $J = \partial F/\partial u$; KINDENSE and KINBAND work with dense and banded approximations to J, respectively. The SUNDIALS suite includes both internal implementations of these two linear solvers and interfaces to Lapack implementations. Together, these linear solvers are referred to as KINDLS (from Direct Linear Solvers).

The second two linear solvers are sparse direct solvers based on Gaussian elimination, and require user-supplied routines to construct the linear system matrix (in the case of Newton's method, this is the Jacobian $J = \partial F/\partial u$) in compressed-sparse-column format. The SUNDIALS suite does not include internal implementations of these solver libraries, instead requiring compilation of SUNDIALS to link with existing installations of these libraries (if either is missing, SUNDIALS will install without the corresponding interface routines). Together, these linear solvers are referred to as KINSLS (from Sparse Linear Solvers).

The remaining KINSOL linear solvers — KINSPGMR, KINSPFGMR, KINSPBCG, and KINSPTFQMR — are Krylov iterative solvers, which use scaled preconditioned GMRES, scaled preconditioned Flexible GMRES, scaled preconditioned Bi-CGStab, and scaled preconditioned TFQMR, respectively. Together, they are referred to as KINSPILS (from Scaled Preconditioned Iterative Linear Solvers).

With any of the Krylov solvers, only right preconditioning is available. For specification of the preconditioner, see the Krylov solver sections within §4.5.4 and §4.6. If preconditioning is done, user-supplied functions define the right preconditioner matrix P, which should approximate the system Jacobian matrix J.

To specify a KINSOL linear solver, after the call to `KINCreate` but before any calls to `KINSol`, the user's program must call one of the functions `KINDense`/`KINLapackDense`, `KINBand`/`KINLapackBand`, `KINKLU`, `KINSuperLUMT`, `KINSpgmr`, `KINSpfgmr`, `KINSpbcg`, or `KINSptfqmr`, as documented below. The first argument passed to these functions is the KINSOL memory pointer returned by `KINCreate`. A call to one of these functions links the main KINSOL nonlinear solver to a linear solver and allows the user to specify parameters which are specific to a particular solver, such as the half-bandwidths in the KINBAND case. The use of each of the linear solvers involves certain constants and possibly some macros, that are likely to be needed in the user code. These are available in the corresponding header file associated with the linear solver, as specified below.

In each case, the linear solver module used by KINSOL is actually built on top of a generic linear system solver, which may be of interest in itself. These generic solvers, denoted DENSE, BAND, KLU, SUPERLUMT, SPGMR, SPFGMR, SPBCG, and SPTFQMR, are described separately in Chapter 8.

`KINDense`

Call `flag = KINDense(kin_mem, N);`

Description The function `KINDense` selects the KINDENSE linear solver and indicates the use of the internal direct dense linear algebra functions.

The user's main program must include the `kinsol_dense.h` header file.

Arguments

`kin_mem`	(`void *`)	pointer to the KINSOL memory block.
`N`	(`long int`)	problem dimension.

Return value The return value `flag` (of type `int`) is one of

`KINDLS_SUCCESS`	The KINDENSE initialization was successful.
`KINDLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINDLS_ILL_INPUT`	The KINDENSE solver is not compatible with the current NVECTOR module.
`KINDLS_MEM_FAIL`	A memory allocation request failed.

Notes The KINDENSE linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are compatible, while NVECTOR_PARALLEL is not.

`KINLapackDense`

Call `flag = KINLapackDense(kin_mem, N);`

Description The function `KINLapackDense` selects the KINDENSE linear solver and indicates the use of Lapack functions.

The user's main program must include the `kinsol_lapack.h` header file.

Arguments

`kin_mem`	(`void *`)	pointer to the KINSOL memory block.
`N`	(`int`)	problem dimension.

Return value The values of the returned `flag` (of type `int`) are identical to those of `KINDense`.

Notes Note that `N` is restricted to be of type `int` here, because of the corresponding type restriction in the Lapack solvers.

`KINBand`

Call `flag = KINBand(kin_mem, N, mupper, mlower);`

Description The function `KINBand` selects the KINBAND linear solver and indicates the use of the internal direct band linear algebra functions.

The user's main program must include the `kinsol_band.h` header file.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`N`	(`long int`) problem dimension.
`mupper`	(`long int`) upper half-bandwidth of the problem Jacobian (or of the approximation of it).
`mlower`	(`long int`) lower half-bandwidth of the problem Jacobian (or of the approximation of it).

Return value The return value `flag` (of type `int`) is one of

`KINDLS_SUCCESS`	The KINBAND initialization was successful.
`KINDLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINDLS_ILL_INPUT`	The KINBAND solver is not compatible with the current NVECTOR module, or one of the Jacobian half-bandwidths is outside its valid range (0 . . . `N`−1).
`KINDLS_MEM_FAIL`	A memory allocation request failed.

Notes The KINBAND linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are compatible, while NVECTOR_PARALLEL is not. The half-bandwidths are to be set so that the nonzero locations (i, j) in the banded (approximate) Jacobian satisfy $-$`mlower` $\leq j - i \leq$ `mupper`.

`KINLapackBand`

Call `flag = KINLapackBand(kin_mem, N, mupper, mlower);`

Description The function `KINLapackBand` selects the KINBAND linear solver and indicates the use of Lapack functions.

The user's main program must include the `kinsol_lapack.h` header file.

Arguments The input arguments are identical to those of `KINBand`, except that `N`, `mupper`, and `mlower` are of type `int` here.

Return value The values of the returned `flag` (of type `int`) are identical to those of `KINBand`.

Notes Note that `N`, `mupper`, and `mlower` are restricted to be of type `int` here, because of the corresponding type restriction in the Lapack solvers.

`KINKLU`

Call `flag = KINKLU(kin_mem, N, NNZ);`

Description The function `KINKLU` selects the KINKLU linear solver and indicates the use of sparse-direct linear algebra functions.

The user's main program must include the `kinsol_klu.h` header file.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`N`	(`int`) problem dimension.
`NNZ`	(`int`) problem dimension.

Return value The return value `flag` (of type `int`) is one of

`KINSLS_SUCCESS`	The KINKLU initialization was successful.
`KINSLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSLS_ILL_INPUT`	The KINKLU solver is not compatible with the current NVECTOR module.
`KINSLS_MEM_FAIL`	A memory allocation request failed.
`KINSLS_PACKAGE_FAIL`	A call to the KLU library returned a failure flag.

Notes	The KINKLU linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are compatible, while NVECTOR_PARALLEL is not.

`KINSuperLUMT`

Call `flag = KINSuperLUMT(kin_mem, num_threads, N, NNZ);`

Description The function `KINSuperLUMT` selects the KINSUPERLUMT linear solver and indicates the use of sparse-direct linear algebra functions.

The user's main program must include the `kinsol_superlumt.h` header file.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`num_threads`	(`int`) the number of threads to use when factoring the linear systems. Note that SuperLU_MT is thread-parallel only in the factorization routine.
`N`	(`int`) problem dimension.
`NNZ`	(`int`) maximum number of nonzero entries in the system Jacobian.

Return value The return value `flag` (of type `int`) is one of

`KINSLS_SUCCESS`	The KINSUPERLUMT initialization was successful.
`KINSLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSLS_ILL_INPUT`	The KINSUPERLUMT solver is not compatible with the current NVECTOR module.
`KINSLS_MEM_FAIL`	A memory allocation request failed.
`KINSLS_PACKAGE_FAIL`	A call to the SuperLU_MT library returned a failure flag.

Notes The KINSUPERLUMT linear solver is not compatible with all implementations of the NVECTOR module. Of the NVECTOR modules provided with SUNDIALS, only NVECTOR_SERIAL, NVECTOR_OPENMP and NVECTOR_PTHREADS are compatible, while NVECTOR_PARALLEL is not.

Performance will significantly degrade if the user applies the SuperLU_MT package compiled with PThreads while using the NVECTOR_OPENMP module. If a user wants to use a threaded vector kernel with this thread-parallel solver, then SuperLU_MT should be compiled with openMP and the NVECTOR_OPENMP module should be used. Also, note that the expected benefit of using the threaded vector kernel is minimal compared to the potential benefit of the threaded solver, unless very long (greater than 100,000 entries) vectors are used.

`KINSpgmr`

Call `flag = KINSpgmr(kin_mem, maxl);`

Description The function `KINSpgmr` selects the KINSPGMR linear solver.

The user's main program must include the `kinsol_spgmr.h` header file.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`maxl`	(`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The KINSPGMR initialization was successful.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_ILL_INPUT`	The NVECTOR module used does not implement a required operation.
`KINSPILS_MEM_FAIL`	A memory allocation request failed.

`KINSpfgmr`

Call	`flag = KINSpfgmr(kin_mem, maxl);`
Description	The function `KINSpfgmr` selects the KINSPFGMR linear solver. The user's main program must include the `kinsol_spfgmr.h` header file.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.
Return value	The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The KINSPFGMR initialization was successful.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_ILL_INPUT`	The NVECTOR module used does not implement a required operation.
`KINSPILS_MEM_FAIL`	A memory allocation request failed.

`KINSpbcg`

Call	`flag = KINSpbcg(kin_mem, maxl);`
Description	The function `KINSpbcg` selects the KINSPBCG linear solver. The user's main program must include the `kinsol_spbcgs.h` header file.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.
Return value	The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The KINSPBCG initialization was successful.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_ILL_INPUT`	The NVECTOR module used does not implement a required operation.
`KINSPILS_MEM_FAIL`	A memory allocation request failed.

`KINSptfqmr`

Call	`flag = KINSptfqmr(kin_mem, maxl);`
Description	The function `KINSptfqmr` selects the KINSPTFQMR linear solver. The user's main program must include the `kinsol_sptfqmr.h` header file.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `maxl` (`int`) maximum dimension of the Krylov subspace to be used. Pass 0 to use the default value `KINSPILS_MAXL`= 5.
Return value	The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The KINSPTFQMR initialization was successful.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_ILL_INPUT`	The NVECTOR module used does not implement a required operation.
`KINSPILS_MEM_FAIL`	A memory allocation request failed.

4.5.3 KINSOL solver function

This is the central step in the solution process, the call to solve the nonlinear algebraic system.

`KINSol`

Call `flag = KINSol(kin_mem, u, strategy, u_scale, f_scale);`

Description The function `KINSol` computes an approximate solution to the nonlinear system.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`u`	(`N_Vector`) vector set to initial guess by user before calling `KINSol`, but which upon return contains an approximate solution of the nonlinear system $F(u) = 0$.
`strategy`	(`int`) strategy used to solve the nonlinear system. It must be of the following: `KIN_NONE` basic Newton iteration `KIN_LINESEARCH` Newton with globalization `KIN_FP` fixed-point iteration with Anderson Acceleration `KIN_PICARD` Picard iteration with Anderson Acceleration
`u_scale`	(`N_Vector`) vector containing diagonal elements of scaling matrix D_u for vector `u` chosen so that the components of D_u·`u` (as a matrix multiplication) all have roughly the same magnitude when `u` is close to a root of $F(u)$.
`f_scale`	(`N_Vector`) vector containing diagonal elements of scaling matrix D_F for $F(u)$ chosen so that the components of $D_F \cdot F($`u`$)$ (as a matrix multiplication) all have roughly the same magnitude when `u` is not too near a root of $F(u)$. In the case of a fixed-point iteration, consider $F(u) = G(u) - u$.

Return value On return, `KINSol` returns the approximate solution in the vector `u` if successful. The return value `flag` (of type `int`) will be one of the following:

`KIN_SUCCESS`

`KINSol` succeeded; the scaled norm of $F(u)$ is less than `fnormtol`.

`KIN_INITIAL_GUESS_OK`

The guess `u` $= u_0$ satisfied the system $F(u) = 0$ within the tolerances specified.

`KIN_STEP_LT_STPTOL`

KINSOL stopped based on scaled step length. This means that the current iterate may be an approximate solution of the given nonlinear system, but it is also quite possible that the algorithm is "stalled" (making insufficient progress) near an invalid solution, or that the scalar `scsteptol` is too large (see `KINSetScaledStepTol` in §4.5.4 to change `scsteptol` from its default value).

`KIN_MEM_NULL`

The KINSOL memory block pointer was `NULL`.

`KIN_ILL_INPUT`

An input parameter was invalid.

`KIN_NO_MALLOC`

The KINSOL memory was not allocated by a call to `KINCreate`.

`KIN_LINESEARCH_NONCONV`

The line search algorithm was unable to find an iterate sufficiently distinct from the current iterate, or could not find an iterate satisfying the sufficient decrease condition.

Failure to satisfy the sufficient decrease condition could mean the current iterate is "close" to an approximate solution of the given nonlinear system, the difference approximation of the matrix-vector product $J(u)v$ is inaccurate, or the real scalar `scsteptol` is too large.

`KIN_MAXITER_REACHED`

The maximum number of nonlinear iterations has been reached.

KIN_MXNEWT_5X_EXCEEDED

Five consecutive steps have been taken that satisfy the inequality $\|D_u p\|_{L2} > 0.99$ `mxnewtstep`, where p denotes the current step and `mxnewtstep` is a scalar upper bound on the scaled step length. Such a failure may mean that $\|D_F F(u)\|_{L2}$ asymptotes from above to a positive value, or the real scalar `mxnewtstep` is too small.

KIN_LINESEARCH_BCFAIL

The line search algorithm was unable to satisfy the "beta-condition" for `MXNBCF` +1 nonlinear iterations (not necessarily consecutive), which may indicate the algorithm is making poor progress.

KIN_LINSOLV_NO_RECOVERY

The user-supplied routine `psolve` encountered a recoverable error, but the preconditioner is already current.

KIN_LINIT_FAIL

The linear solver initialization routine (`linit`) encountered an error.

KIN_LSETUP_FAIL

The user-supplied routine `pset` (used to set up the preconditioner data) encountered an unrecoverable error.

KIN_LSOLVE_FAIL

Either the user-supplied routine `psolve` (used to to solve the preconditioned linear system) encountered an unrecoverable error, or the linear solver routine (`lsolve`) encountered an error condition.

KIN_SYSFUNC_FAIL

The system function failed in an unrecoverable manner.

KIN_FIRST_SYSFUNC_ERR

The system function failed recoverably at the first call.

KIN_REPTD_SYSFUNC_ERR

The system function had repeated recoverable errors. No recovery is possible.

Notes The components of vectors `u_scale` and `f_scale` should be strictly positive.

KIN_SUCCESS = 0, KIN_INITIAL_GUESS_OK = 1, and KIN_STEP_LT_STPTOL = 2. All remaining return values are negative and therefore a test `flag` < 0 will trap all `KINSol` failures.

4.5.4 Optional input functions

There are numerous optional input parameters that control the behavior of the KINSOL solver. KINSOL provides functions that can be used to change these from their default values. Table 4.1 lists all optional input functions in KINSOL which are then described in detail in the remainder of this section, beginning with those for the main KINSOL solver and continuing with those for the linear solver modules. For the most casual use of KINSOL, the reader can skip to §4.6.

We note that, on error return, all of these functions also send an error message to the error handler function. We also note that all error return values are negative, so a test `flag` < 0 will catch any error.

4.5.4.1 Main solver optional input functions

The calls listed here can be executed in any order. However, if either of the functions `KINSetErrFile` or `KINSetErrHandlerFn` is to be called, that call should be first, in order to take effect for any later error message.

Table 4.1: Optional inputs for KINSOL, KINDENSE, KINSPARSE, and KINSPILS

Optional input	**Function name**	**Default**
	KINSOL main solver	
Error handler function	`KINSetErrHandlerFn`	internal fn.
Pointer to an error file	`KINSetErrFile`	`stderr`
Info handler function	`KINSetInfoHandlerFn`	internal fn.
Pointer to an info file	`KINSetInfoFile`	`stdout`
Data for problem-defining function	`KINSetUserData`	`NULL`
Verbosity level of output	`KINSetPrintLevel`	0
Max. number of nonlinear iterations	`KINSetNumMaxIters`	200
No initial matrix setup	`KINSetNoInitSetup`	`FALSE`
No residual monitoring*	`KINSetNoResMon`	`FALSE`
Max. iterations without matrix setup	`KINSetMaxSetupCalls`	10
Max. iterations without residual check*	`KINSetMaxSubSetupCalls`	5
Form of η coefficient	`KINSetEtaForm`	`KIN_ETACHOICE1`
Constant value of η	`KINSetEtaConstValue`	0.1
Values of γ and α	`KINSetEtaParams`	0.9 and 2.0
Values of ω_{min} and ω_{max}*	`KINSetResMonParams`	0.00001 and 0.9
Constant value of ω*	`KINSetResMonConstValue`	0.9
Lower bound on ϵ	`KINSetNoMinEps`	`FALSE`
Max. scaled length of Newton step	`KINSetMaxNewtonStep`	$1000\|D_u u_0\|_2$
Max. number of β-condition failures	`KINSetMaxBetaFails`	10
Rel. error for D.Q. Jv	`KINSetRelErrFunc`	$\sqrt{\text{uround}}$
Function-norm stopping tolerance	`KINSetFuncNormTol`	$\text{uround}^{1/3}$
Scaled-step stopping tolerance	`KINSetScaledSteptol`	$\text{uround}^{2/3}$
Inequality constraints on solution	`KINSetConstraints`	`NULL`
Nonlinear system function	`KINSetSysFunc`	none
Anderson Acceleration subspace size	`KINSetMAA`	0
	KINDLS linear solvers	
Dense Jacobian function	`KINDlsSetDenseJacFn`	DQ
Band Jacobian function	`KINDlsSetBandJacFn`	DQ
	KINSLS linear solvers	
Sparse Jacobian function	`KINSlsSetSparseJacFn`	none
Sparse matrix ordering algorithm	`KINKLUSetOrdering`	1 for `COLAMD`
Sparse matrix ordering algorithm	`KINSuperLUMTSetOrdering`	3 for `COLAMD`
	KINSPILS linear solvers	
Max. number of restarts**	`KINSpilsSetMaxRestarts`	0
Preconditioner functions and data	`KINSpilsSetPreconditioner`	`NULL`, `NULL`, `NULL`
Jacobian-times-vector function and data	`KINSpilsSetJacTimesVecFn`	internal DQ, `NULL`

* Only for the KINDLS linear solvers
** Only for KINSPGMR and KINSPFGMR

`KINSetErrFile`

Call `flag = KINSetErrFile(kin_mem, errfp);`

Description The function `KINSetErrFile` specifies the pointer to the file where all KINSOL messages should be directed when the default KINSOL error handler function is used.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`errfp` (`FILE *`) pointer to output file.

Return value The return value `flag` (of type `int`) is one of

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

Notes The default value for `errfp` is `stderr`.

Passing a value of `NULL` disables all future error message output (except for the case in which the KINSOL memory pointer is `NULL`). This use of `KINSetErrFile` is strongly discouraged.

! If `KINSetErrFile` is to be called, it should be called before any other optional input functions, in order to take effect for any later error message.

`KINSetErrHandlerFn`

Call `flag = KINSetErrHandlerFn(kin_mem, ehfun, eh_data);`

Description The function `KINSetErrHandlerFn` specifies the optional user-defined function to be used in handling error messages.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`ehfun` (`KINErrHandlerFn`) is the user's C error handler function (see §4.6.2).
`eh_data` (`void *`) pointer to user data passed to `ehfun` every time it is called.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The function `ehfun` and data pointer `eh_data` have been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

Notes The default internal error handler function directs error messages to the file specified by the file pointer `errfp` (see `KINSetErrFile` above).

Error messages indicating that the KINSOL solver memory is `NULL` will always be directed to `stderr`.

`KINSetInfoFile`

Call `flag = KINSetInfoFile(kin_mem, infofp);`

Description The function `KINSetInfoFile` specifies the pointer to the file where all informative (non-error) messages should be directed.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`infofp` (`FILE *`) pointer to output file.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

Notes The default value for `infofp` is `stdout`.

`KINSetInfoHandlerFn`

Call	`flag = KINSetInfoHandlerFn(kin_mem, ihfun, ih_data);`
Description	The function `KINSetInfoHandlerFn` specifies the optional user-defined function to be used in handling informative (non-error) messages.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `ihfun` (`KINInfoHandlerFn`) is the user's C information handler function (see §4.6.3). `ih_data` (`void *`) pointer to user data passed to `ihfun` every time it is called.
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The function `ihfun` and data pointer `ih_data` have been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
Notes	The default internal information handler function directs informative (non-error) messages to the file specified by the file pointer `infofp` (see `KINSetInfoFile` above).

`KINSetPrintLevel`

Call `flag = KINSetPrintLevel(kin_mem, printfl);`

Description The function `KINSetPrintLevel` specifies the level of verbosity of the output.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`printfl` (`int`) flag indicating the level of verbosity. Must be one of:

0 no information displayed.

1 for each nonlinear iteration display the following information: the scaled Euclidean ℓ_2 norm of the system function evaluated at the current iterate, the scaled norm of the Newton step (only if using `KIN_NONE`), and the number of function evaluations performed so far.

2 display level 1 output and the following values for each iteration:
$\|F(u)\|_{D_F}$ (only for `KIN_NONE`).
$\|F(u)\|_{D_F,\infty}$ (for `KIN_NONE` and `KIN_LINESEARCH`).

3 display level 2 output plus additional values used by the global strategy (only if using `KIN_LINESEARCH`), and statistical information for the linear solver.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The argument `printfl` had an illegal value.

Notes The default value for `printfl` is 0.

`KINSetUserData`

Call `flag = KINSetUserData(kin_mem, user_data);`

Description The function `KINSetUserData` specifies the pointer to user-defined memory that is to be passed to all user-supplied functions.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`user_data` (`void *`) pointer to the user-defined memory.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

Notes	If specified, the pointer to `user_data` is passed to all user-supplied functions that have it as an argument. Otherwise, a `NULL` pointer is passed. If `user_data` is needed in user preconditioner functions, the call to `KINSetUserData` must be made *before* the call to specify the linear solver.

`KINSetNumMaxIters`

Call	`flag = KINSetNumMaxIters(kin_mem, mxiter);`
Description	The function `KINSetNumMaxIters` specifies the maximum number of nonlinear iterations allowed.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `mxiter` (`long int`) maximum number of nonlinear iterations.
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional value has been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`. `KIN_ILL_INPUT` The maximum number of iterations was non-positive.
Notes	The default value for `mxiter` is `MXITER_DEFAULT` = 200.

`KINSetNoInitSetup`

Call	`flag = KINSetNoInitSetup(kin_mem, noInitSetup);`
Description	The function `KINSetNoInitSetup` specifies whether an initial call to the preconditioner or Jacobian setup function should be made or not.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `noInitSetup` (`booleantype`) flag controlling whether an initial call to the preconditioner or Jacobian setup function is made (pass `FALSE`) or not made (pass `TRUE`).
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional value has been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
Notes	The default value for `noInitSetup` is `FALSE`, meaning that an initial call to the preconditioner or Jacobian setup function will be made. A call to this function is useful when solving a sequence of problems, in which the final preconditioner or Jacobian value from one problem is to be used initially for the next problem.

`KINSetNoResMon`

Call	`flag = KINSetNoResMon(kin_mem, noNNIResMon);`
Description	The function `KINSetNoResMon` specifies whether or not the nonlinear residual monitoring scheme is used to control Jacobian updating
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `noNNIResMon` (`booleantype`) flag controlling whether residual monitoring is used (pass `FALSE`) or not used (pass `TRUE`).
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional value has been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

Notes When using a direct solver, the default value for `noNNIResMon` is `FALSE`, meaning that the nonlinear residual will be monitored.

! Residual monitoring is only available for use with the direct linear solver modules (meaning KINDENSE, KINBAND, KINKLU, and KINSUPERLUMT).

`KINSetMaxSetupCalls`

Call `flag = KINSetMaxSetupCalls(kin_mem, msbset);`

Description The function `KINSetMaxSetupCalls` specifies the maximum number of nonlinear iterations that can be performed between calls to the preconditioner or Jacobian setup function.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`msbset` (`long int`) maximum number of nonlinear iterations without a call to the preconditioner or Jacobian setup function. Pass 0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The argument `msbset` was negative.

Notes The default value for `msbset` is `MSBSET_DEFAULT` = 10.

`KINSetMaxSubSetupCalls`

Call `flag = KINSetMaxSubSetupCalls(kin_mem, msbsetsub);`

Description The function `KINSetMaxSubSetupCalls` specifies the maximum number of nonlinear iterations between checks by the residual monitoring algorithm.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`msbsetsub` (`long int`) maximum number of nonlinear iterations without checking the nonlinear residual. Pass 0 to indicate the default.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The argument `msbsetsub` was negative.

Notes The default value for `msbsetsub` is `MSBSET_SUB_DEFAULT` = 5.

Residual monitoring is only available for use with the direct linear solver modules (meaning KINDENSE, KINBAND, KINKLU, and KINSUPERLUMT). !

`KINSetEtaForm`

Call `flag = KINSetEtaForm(kin_mem, etachoice);`

Description The function `KINSetEtaForm` specifies the method for computing the value of the η coefficient used in the calculation of the linear solver convergence tolerance.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`etachoice` (`int`) flag indicating the method for computing η. The value must be one of `KIN_ETACHOICE1`, `KIN_ETACHOICE2`, or `KIN_ETACONSTANT` (see Chapter 2 for details).

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

`KIN_ILL_INPUT`	The argument `etachoice` had an illegal value.

Notes: The default value for `etachoice` is `KIN_ETACHOICE1`.

`KINSetEtaConstValue`

Call: `flag = KINSetEtaConstValue(kin_mem, eta);`

Description: The function `KINSetEtaConstValue` specifies the constant value for η in the case `etachoice = KIN_ETACONSTANT`.

Arguments:

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`eta`	(`realtype`) constant value for η. Pass 0.0 to indicate the default.

Return value: The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The argument `eta` had an illegal value

Notes: The default value for `eta` is 0.1. The legal values are $0.0 <$ `eta` ≤ 1.0.

`KINSetEtaParams`

Call: `flag = KINSetEtaParams(kin_mem, egamma, ealpha);`

Description: The function `KINSetEtaParams` specifies the parameters γ and α in the formula for η, in the case `etachoice = KIN_ETACHOICE2`.

Arguments:

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`egamma`	(`realtype`) value of the γ parameter. Pass 0.0 to indicate the default.
`ealpha`	(`realtype`) value of the α parameter. Pass 0.0 to indicate the default.

Return value: The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional values have been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	One of the arguments `egamma` or `ealpha` had an illegal value.

Notes: The default values for `egamma` and `ealpha` are 0.9 and 2.0, respectively.

The legal values are $0.0 <$ `egamma` ≤ 1.0 and $1.0 <$ `ealpha` ≤ 2.0.

`KINSetResMonConstValue`

Call: `flag = KINSetResMonConstValue(kin_mem, omegaconst);`

Description: The function `KINSetResMonConstValue` specifies the constant value for ω when using residual monitoring.

Arguments:

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`omegaconst`	(`realtype`) constant value for ω. Passing 0.0 results in using Eqn. (2.4).

Return value: The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The argument `omegaconst` had an illegal value

Notes: The default value for `omegaconst` is 0.9. The legal values are $0.0 <$ `omegaconst` < 1.0.

`KINSetResMonParams`

Call	`flag = KINSetResMonParams(kin_mem, omegamin, omegamax);`
Description	The function `KINSetResMonParams` specifies the parameters ω_{min} and ω_{max} in the formula (2.4) for ω.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `omegamin` (`realtype`) value of the ω_{min} parameter. Pass 0.0 to indicate the default. `omegamax` (`realtype`) value of the ω_{max} parameter. Pass 0.0 to indicate the default.
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional values have been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`. `KIN_ILL_INPUT` One of the arguments `omegamin` or `omegamax` had an illegal value.
Notes	The default values for `omegamin` and `omegamax` are 0.00001 and 0.9, respectively. The legal values are $0.0 <$ `omegamin` $<$ `omegamax` < 1.0.

`KINSetNoMinEps`

Call	`flag = KINSetNoMinEps(kin_mem, noMinEps);`
Description	The function `KINSetNoMinEps` specifies a flag that controls whether or not the value of ϵ, the scaled linear residual tolerance, is bounded from below.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `noMinEps` (`booleantype`) flag controlling the bound on ϵ.
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional value has been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.
Notes	The default value for `noMinEps` is `FALSE`, meaning that a positive minimum value, equal to 0.01*`fnormtol`, is applied to ϵ. (See `KINSetFuncNormTol` below.)

`KINSetMaxNewtonStep`

Call	`flag = KINSetMaxNewtonStep(kin_mem, mxnewtstep);`
Description	The function `KINSetMaxNewtonStep` specifies the maximum allowable scaled length of the Newton step.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `mxnewtstep` (`realtype`) maximum scaled step length (≥ 0.0). Pass 0.0 to indicate the default.
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional value has been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`. `KIN_ILL_INPUT` The input value was negative.
Notes	The default value of `mxnewtstep` is $1000\,\|u_0\|_{D_u}$, where u_0 is the initial guess.

`KINSetMaxBetaFails`

Call	`flag = KINSetMaxBetaFails(kin_mem, mxnbcf);`
Description	The function `KINSetMaxBetaFails` specifies the maximum number of β-condition failures in the linesearch algorithm.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block.

`mxnbcf`	(`realtype`) maximum number of β-condition failures. Pass 0.0 to indicate the default.

Return value: The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	`mxnbcf` was negative.

Notes: The default value of `mxnbcf` is `MXNBCF_DEFAULT` $= 10$.

`KINSetRelErrFunc`

Call: `flag = KINSetRelErrFunc(kin_mem, relfunc);`

Description: The function `KINSetRelErrFunc` specifies the relative error in computing $F(u)$, which is used in the difference quotient approximation to the Jacobian matrix [see Eq.(2.7)] or the Jacobian-vector product [see Eq.(2.9)].

Arguments:

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`relfunc`	(`realtype`) relative error in $F(u)$ (`relfunc` ≥ 0.0). Pass 0.0 to indicate the default.

Return value: The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The relative error was negative.

Notes: The default value for `relfunc` is U = unit roundoff.

`KINSetFuncNormTol`

Call: `flag = KINSetFuncNormTol(kin_mem, fnormtol);`

Description: The function `KINSetFuncNormTol` specifies the scalar used as a stopping tolerance on the scaled maximum norm of the system function $F(u)$.

Arguments:

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`fnormtol`	(`realtype`) tolerance for stopping based on scaled function norm (≥ 0.0). Pass 0.0 to indicate the default.

Return value: The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The tolerance was negative.

Notes: The default value for `fnormtol` is (unit roundoff)$^{1/3}$.

`KINSetScaledStepTol`

Call: `flag = KINSetScaledStepTol(kin_mem, scsteptol);`

Description: The function `KINSetScaledStepTol` specifies the scalar used as a stopping tolerance on the minimum scaled step length.

Arguments:

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`scsteptol`	(`realtype`) tolerance for stopping based on scaled step length (≥ 0.0). Pass 0.0 to indicate the default.

Return value: The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.

KIN_MEM_NULL	The kin_mem pointer is NULL.
KIN_ILL_INPUT	The tolerance was non-positive.

Notes The default value for scsteptol is (unit roundoff)$^{2/3}$.

`KINSetConstraints`

Call `flag = KINSetConstraints(kin_mem, constraints);`

Description The function KINSetConstraints specifies a vector that defines inequality constraints for each component of the solution vector u.

Arguments

kin_mem	(void *) pointer to the KINSOL memory block.
constraints	(N_Vector) vector of constraint flags. If constraints[i] is

0.0	then no constraint is imposed on u_i.
1.0	then u_i will be constrained to be $u_i \geq 0.0$.
-1.0	then u_i will be constrained to be $u_i \leq 0.0$.
2.0	then u_i will be constrained to be $u_i > 0.0$.
-2.0	then u_i will be constrained to be $u_i < 0.0$.

Return value The return value flag (of type int) is one of:

KIN_SUCCESS	The optional value has been successfully set.
KIN_MEM_NULL	The kin_mem pointer is NULL.
KIN_ILL_INPUT	The constraint vector contains illegal values.

Notes The presence of a non-NULL constraints vector that is not 0.0 in all components will cause constraint checking to be performed.

The function creates a private copy of the constraints vector. Consequently, the user-supplied vector can be freed after the function call, and the constraints can only be changed by calling this function.

`KINSetSysFunc`

Call `flag = KINSetSysFunc(kin_mem, func);`

Description The function KINSetSysFunc specifies the user-provided function that evaluates the nonlinear system function $F(u)$ or $G(u)$.

Arguments

kin_mem	(void *) pointer to the KINSOL memory block.
func	(KINSysFn) user-supplied function that evaluates $F(u)$ (or $G(u)$ for fixed-point iteration).

Return value The return value flag (of type int) is one of:

KIN_SUCCESS	The optional value has been successfully set.
KIN_MEM_NULL	The kin_mem pointer is NULL.
KIN_ILL_INPUT	The argument func was NULL.

Notes The nonlinear system function is initially specified through KINInit. The option of changing the system function is provided for a user who wishes to solve several problems of the same size but with different functions.

`KINSetMAA`

Call `flag = KINSetMAA(kin_mem, maa);`

Description The function KINSetMAA specifies the size of the subspace used with Anderson acceleration in conjunction with Picard or fixed-point iteration.

Arguments kin_mem (void *) pointer to the KINSOL memory block.

`maa`	(`long int`) subspace size for various methods. A value of 0 means no acceleration, while a positive value means acceleration will be done.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KIN_ILL_INPUT`	The argument `maa` was negative.

Notes This function sets the subspace size, which needs to be > 0 if Anderson Acceleration is to be used. It also allocates additional memory necessary for Anderson Acceleration.

The default value of `maa` is 0, indicating no acceleration. The value of `maa` should always be less than `mxiter`.

If the user calls the function KINSetNumMaxIters, that call should be made before the call to KINSetMAA, as the latter uses the value of `mxiter`.

4.5.4.2 Dense/band direct linear solver optional input functions

The KINDENSE solver needs a function to compute a dense approximation to the Jacobian matrix $J(u)$. This function must be of type `KINDlsDenseJacFn`. The user can supply his/her own dense Jacobian function, or use the default internal difference quotient approximation that comes with the KINDENSE solver. To specify a user-supplied Jacobian function `djac`, KINDENSE provides the function `KINDlsSetDenseJacFn`. The KINDENSE solver passes the pointer `user_data` to the dense Jacobian function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `user_data` may be specified through `KINSetUserData`.

`KINDlsSetDenseJacFn`

Call `flag = KINDlsSetDenseJacFn(kin_mem, djac);`

Description The function `KINDlsSetDenseJacFn` specifies the dense Jacobian approximation function to be used.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`djac`	(`KINDlsDenseJacFn`) user-defined dense Jacobian approximation function.

Return value The return value `flag` (of type `int`) is one of

`KINDLS_SUCCESS`	The optional value has been successfully set.
`KINDLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINDLS_LMEM_NULL`	The KINDENSE linear solver has not been initialized.

Notes By default, KINDENSE uses an internal difference quotient function. If `NULL` is passed to `djac`, this default function is used.

The function type `KINDlsDenseJacFn` is described in §4.6.4.

The KINBAND solver needs a function to compute a banded approximation to the Jacobian matrix $J(u)$. This function must be of type `KINDlsBandJacFn`. The user can supply his/her own banded Jacobian approximation function, or use the default internal difference quotient approximation that comes with the KINBAND solver. To specify a user-supplied Jacobian function `bjac` KINBAND provides the function `KINDlsSetBandJacFn`. The KINBAND solver passes the pointer `user_data` to the banded Jacobian approximation function. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `user_data` may be specified through `KINSetUserData`.

`KINDlsSetBandJacFn`

Call	`flag = KINDlsSetBandJacFn(kin_mem, bjac);`
Description	The function `KINBandSetJacFn` specifies the banded Jacobian approximation function to be used.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `bjac` (`KINDlsBandJacFn`) user-defined banded Jacobian approximation function.
Return value	The return value `flag` (of type `int`) is one of `KINDLS_SUCCESS` The optional value has been successfully set. `KINDLS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINDLS_LMEM_NULL` The KINBAND linear solver has not been initialized.
Notes	By default, KINBAND uses an internal difference quotient approximation. If `NULL` is passed to `bjac`, this default function is used. The function type `KINDlsBandJacFn` is described in §4.6.5.

4.5.4.3 Sparse linear solvers optional input functions

The KINKLU and KINSUPERLUMT solvers require a function to compute a compressed-sparse-column approximation to the Jacobian matrix $J(u)$. This function must be of type `KINSlsSparseJacFn`. The user must supply a custom sparse Jacobian function since a difference-quotient approximation would not leverage the underlying sparse matrix structure of the problem. To specify a user-supplied Jacobian function `sjac`, KINKLU and KINSUPERLUMT provide the function `KINSlsSetSparseJacFn`. The KINKLU and KINSUPERLUMT solvers pass the pointer `user_data` to the sparse Jacobian function. This mechanism allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied Jacobian function, without using global data in the program. The pointer `user_data` may be specified through `KINSetUserData`.

`KINSlsSetSparseJacFn`

Call	`flag = KINSlsSetSparseJacFn(kin_mem, sjac);`
Description	The function `KINSlsSetSparseJacFn` specifies the sparse Jacobian approximation function to be used.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `sjac` (`KINSlsSparseJacFn`) user-defined sparse Jacobian approximation function.
Return value	The return value `flag` (of type `int`) is one of `KINSLS_SUCCESS` The optional value has been successfully set. `KINSLS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSLS_LMEM_NULL` The KINSLS linear solver has not been initialized.
Notes	The function type `KINSlsSparseJacFn` is described in §4.6.6.

When using a sparse direct solver, there may be instances when the number of state variables does not change, but the number of nonzeroes in the Jacobian does change. In this case, for the KINKLU solver, we provide the following reinitialization function. This function reinitializes the Jacobian matrix memory for the new number of nonzeroes and sets flags for a new factorization (symbolic and numeric) to be conducted at the next solver setup call. This routine is useful in the cases where the number of nonzeroes has changed, or where the structure of the linear system has changed, requiring a new symbolic (and numeric) factorization.

`KINKLUReInit`

Call `flag = KINKLUReInit(kin_mem, n, nnz, reinit_type);`

Description The function `KINKLUReInit` reinitializes Jacobian matrix memory and flags for new symbolic and numeric KLU factorizations.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`n` (`int`) number of state variables in the system.
`nnz` (`int`) number of nonzeroes in the Jacobian matrix.
`reinit_type` (`int`) type of reinitialization:

1 The Jacobian matrix will be destroyed and a new one will be allocated based on the `nnz` value passed to this call. New symbolic and numeric factorizations will be completed at the next solver setup.

2 Only symbolic and numeric factorizations will be completed. It is assumed that the Jacobian size has not exceeded the size of `nnz` given in the prior call to KINKLU.

Return value The return value `flag` (of type `int`) is one of

`KINSLS_SUCCESS`	The reinitialization succeeded.
`KINSLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSLS_LMEM_NULL`	The KINKLU linear solver has not been initialized.
`KINSLS_ILL_INPUT`	The given `reinit_type` has an illegal value.
`KINSLS_MEM_FAIL`	A memory allocation failed.

Notes The default value for `reinit_type` is 2.

Both the KINKLU and KINSUPERLUMT solvers can apply reordering algorithms to minimize fill-in for the resulting sparse *LU* decomposition internal to the solver. The approximate minimal degree ordering for nonsymmetric matrices given by the `COLAMD` algorithm is the default algorithm used within both solvers, but alternate orderings may be chosen through one of the following two functions. The input values to these functinos are the numeric values used in the respective packages, and the user-supplied value will be passed directly to the package.

`KINKLUSetOrdering`

Call `flag = KINKLUSetOrdering(kin_mem, ordering_choice);`

Description The function `KINKLUSetOrdering` specifies the ordering algorithm used by KINKLU for reducing fill.

Arguments
`kin_mem` (`void *`) pointer to the KINSOL memory block.
`ordering_choice` (`int`) flag denoting algorithm choice:

0 `AMD`
1 `COLAMD`
2 natural ordering

Return value The return value `flag` (of type `int`) is one of

`KINSLS_SUCCESS`	The optional value has been successfully set.
`KINSLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSLS_ILL_INPUT`	The supplied value of `ordering_choice` is illegal.

Notes The default ordering choice is 1 for `COLAMD`.

`KINSuperLUMTSetOrdering`

Call	`flag = KINSuperLUMTSetOrdering(kin_mem, ordering_choice);`
Description	The function `KINSuperLUMTSetOrdering` specifies the ordering algorithm used by KINSUPERLUMT for reducing fill.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `ordering_choice` (`int`) flag denoting algorithm choice: 0 natural ordering 1 minimal degree ordering on $J^T J$ 2 minimal degree ordering on $J^T + J$ 3 `COLAMD`
Return value	The return value `flag` (of type `int`) is one of `KINSLS_SUCCESS` The optional value has been successfully set. `KINSLS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSLS_ILL_INPUT` The supplied value of `ordering_choice` is illegal.
Notes	The default ordering choice is 3 for `COLAMD`.

4.5.4.4 Iterative linear solvers optional input functions

If any preconditioning is to be done with one of the KINSPILS linear solvers, then the user must supply a preconditioner solve function `psolve` and specify its name in a call to `KINSpilsSetPreconditioner`.

The evaluation and preprocessing of any Jacobian-related data needed by the user's preconditioner solve function is done in the optional user-supplied function `psetup`. Both of these functions are fully specified in §4.6. If used, the `psetup` function should also be specified in the call to `KINSpilsSetPreconditioner`. A KINSPILS solver passes the pointer `user_data` received through `KINSetUserData` to the preconditioner `psetup` and `psolve` functions. This allows the user to create an arbitrary structure with relevant problem data and access it during the execution of the user-supplied preconditioner functions without using global data in the program.

Ther KINSPILS solvers require a function to compute an approximation to the product between the Jacobian matrix $J(u)$ and a vector v. The user can supply his/her own Jacobian-times-vector approximation function, or use the internal difference quotient approximation that comes with the KINSPILS solvers. A user-defined Jacobian-vector function must be of type `KINSpilsJacTimesVecFn` and can be specified through a call to `KINSpilsSetJacTimesVecFn` (see §4.6.7 for specification details). A KINSPILS solver passes the pointer `user_data` received through `KINSetUserData` to the Jacobian-times-vector function `jtimes` each time it is called.

`KINSpilsSetPreconditioner`

Call	`flag = KINSpilsSetPreconditioner(kin_mem, psetup, psolve);`
Description	The function `KINSpilsSetPreconditioner` specifies the preconditioner setup and solve functions.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `psetup` (`KINSpilsPrecSetupFn`) user-defined preconditioner setup function. Pass `NULL` if no setup operation is to be done. `psolve` (`KINSpilsPrecSolveFn`) user-defined preconditioner solve function.
Return value	The return value `flag` (of type `int`) is one of `KINSPILS_SUCCESS` The optional values have been successfully set. `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSPILS_LMEM_NULL` The KINSPILS linear solver has not been initialized.
Notes	The function type `KINSpilsPrecSolveFn` is described in §4.6.8. The function type `KINSpilsPrecSetupFn` is described in §4.6.9.

`KINSpilsSetJacTimesVecFn`

Call	`flag = KINSpilsSetJacTimesVecFn(kin_mem, jtimes);`
Description	The function `KINSpilsSetJacTimesFn` specifies the Jacobian-vector function to be used.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `jtimes` (`KINSpilsJacTimesVecFn`) user-defined Jacobian-vector product function.
Return value	The return value `flag` (of type `int`) is one of `KINSPILS_SUCCESS` The optional value has been successfully set. `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSPILS_LMEM_NULL` The KINSPILS linear solver has not been initialized.
Notes	By default, the KINSPILS linear solvers use an internal difference quotient function `KINSpilsDQJtimes`. If `NULL` is passed as `jtimes`, this default function is used. The function type `KINSpilsJacTimesVecFn` is described in §4.6.7.

`KINSpilsSetMaxRestarts`

Call	`flag = KINSpilsSetMaxRestarts(kin_mem, maxrs);`
Description	The function `KINSpilsSetMaxRestarts` specifies the maximum number of times the iterative linear solver can be restarted.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `maxrs` (`int`) maximum number of restarts (≥ 0).
Return value	The return value `flag` (of type `int`) is one of: `KINSPILS_SUCCESS` The optional value has been successfully set. `KINSPILS_ILL_INPUT` The maximum number of restarts specified is negative. `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSPILS_LMEM_NULL` The linear solver has not been initialized.
Notes	The default value is 0 (meaning no restarts). ! This option is available only for the KINSPGMR and KINSPFGMR linear solvers.

4.5.5 Optional output functions

KINSOL provides an extensive list of functions that can be used to obtain solver performance information. Table 4.2 lists all optional output functions in KINSOL, which are then described in detail in the remainder of this section, beginning with those for the main KINSOL solver and continuing with those for the linear solver modules. Where the name of an output from a linear solver module would otherwise conflict with the name of an optional output from the main solver, a suffix `LS` (for Linear Solver) has been added here (*e.g.*, `lenrwLS`).

4.5.5.1 Main solver optional output functions

KINSOL provides several user-callable functions that can be used to obtain different quantities that may be of interest to the user, such as solver workspace requirements and solver performance statistics. These optional output functions are described next.

`KINGetWorkSpace`

Call	`flag = KINGetWorkSpace(kin_mem, &lenrw, &leniw);`
Description	The function `KINGetWorkSpace` returns the KINSOL integer and real workspace sizes.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block.

Table 4.2: Optional outputs from KINSOL, KINDLS, KINSLS, and KINSPILS

Optional output	**Function name**
KINSOL main solver	
Size of KINSOL real and integer workspaces	`KINGetWorkSpace`
Number of function evaluations	`KINGetNumFuncEvals`
Number of nonlinear iterations	`KINGetNumNolinSolvIters`
Number of β-condition failures	`KINGetNumBetaCondFails`
Number of backtrack operations	`KINGetNumBacktrackOps`
Scaled norm of F	`KINGetFuncNorm`
Scaled norm of the step	`KINGetStepLength`
KINDLS linear solvers	
Size of real and integer workspaces	`KINDlsGetWorkSpace`
No. of Jacobian evaluations	`KINDlsGetNumJacEvals`
No. of F calls for D.Q. Jacobian evals.	`KINDlsGetNumFuncEvals`
Last return from a KINDLS function	`KINDlsGetLastFlag`
KINSLS linear solvers	
No. of Jacobian evaluations	`KINSlsGetNumJacEvals`
Last return from a linear solver function	`KINSlsGetLastFlag`
Name of constant associated with a return flag	`KINSlsGetReturnFlagName`
KINSPILS linear solvers	
Size of real and integer workspaces	`KINSpilsGetWorkSpace`
No. of linear iterations	`KINSpilsGetNumLinIters`
No. of linear convergence failures	`KINSpilsGetNumConvFails`
No. of preconditioner evaluations	`KINSpilsGetNumPrecEvals`
No. of preconditioner solves	`KINSpilsGetNumPrecSolves`
No. of Jacobian-vector product evaluations	`KINSpilsGetNumJtimesEvals`
No. of F calls for D.Q. Jacobian-vector evals.	`KINSpilsGetNumFuncEvals`
Last return from a linear solver function	`KINSpilsGetLastFlag`

`lenrw`	(`long int`) the number of `realtype` values in the KINSOL workspace.
`leniw`	(`long int`) the number of integer values in the KINSOL workspace.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional output values have been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

Notes In terms of the problem size N, the actual size of the real workspace is $17+5N$ `realtype` words. The real workspace is increased by an additional N words if constraint checking is enabled (see `KINSetConstraints`).

The actual size of the integer workspace (without distinction between `int` and `long int`) is $22+5N$ (increased by N if constraint checking is enabled).

`KINGetNumFuncEvals`

Call `flag = KINGetNumFuncEvals(kin_mem, &nfevals);`

Description The function `KINGetNumFuncEvals` returns the number of evaluations of the system function.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`nfevals`	(`long int`) number of calls to the user-supplied function that evaluates $F(u)$.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional output value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

`KINGetNumNonlinSolvIters`

Call `flag = KINGetNumNonlinSolvIters(kin_mem, &nniters);`

Description The function `KINGetNumNonlinSolvIters` returns the number of nonlinear iterations.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`nniters`	(`long int`) number of nonlinear iterations.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional output value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

`KINGetNumBetaCondFails`

Call `flag = KINGetNumBetaCondFails(kin_mem, &nbcfails);`

Description The function `KINGetNumBetaCondFails` returns the number of β-condition failures.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`nbcfails`	(`long int`) number of β-condition failures.

Return value The return value `flag` (of type `int`) is one of:

`KIN_SUCCESS`	The optional output value has been successfully set.
`KIN_MEM_NULL`	The `kin_mem` pointer is `NULL`.

`KINGetNumBacktrackOps`

Call	`flag = KINGetNumBacktrackOps(kin_mem, &nbacktr);`
Description	The function `KINGetNumBacktrackOps` returns the number of backtrack operations (step length adjustments) performed by the line search algorithm.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `nbacktr` (`long int`) number of backtrack operations.
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional output value has been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KINGetFuncNorm`

Call	`flag = KINGetFuncNorm(kin_mem, &fnorm);`
Description	The function `KINGetFuncNorm` returns the scaled Euclidean ℓ_2 norm of the nonlinear system function $F(u)$ evaluated at the current iterate.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `fnorm` (`realtype`) current scaled norm of $F(u)$.
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional output value has been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

`KINGetStepLength`

Call	`flag = KINGetStepLength(kin_mem, &steplength);`
Description	The function `KINGetStepLength` returns the scaled Euclidean ℓ_2 norm of the step used during the previous iteration.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `steplength` (`realtype`) scaled norm of the Newton step.
Return value	The return value `flag` (of type `int`) is one of: `KIN_SUCCESS` The optional output value has been successfully set. `KIN_MEM_NULL` The `kin_mem` pointer is `NULL`.

4.5.5.2 Dense/band direct linear solvers optional output functions

The following optional outputs are available from the KINDLS module: workspace requirements, number of calls to the Jacobian routine, number of calls to the system function routine for difference quotient Jacobian approximation, and last return value from a KINDLS function.

`KINDlsGetWorkSpace`

Call	`flag = KINDlsGetWorkSpace(kin_mem, &lenrwLS, &leniwLS);`
Description	The function `KINDlsGetWorkSpace` returns the KINDENSE real and integer workspace sizes.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `lenrwLS` (`long int`) the number of `realtype` values in the KINDLS workspace. `leniwLS` (`long int`) the number of integer values in the KINDLS workspace.
Return value	The return value `flag` (of type `int`) is one of `KINDLS_SUCCESS` The optional output value has been successfully set.

`KINDLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINDLS_LMEM_NULL`	The KINDENSE linear solver has not been initialized.

Notes: For the KINDENSE linear soler, in terms of the problem size N, the actual size of the real workspace is N^2 `realtype` words, and the actual size of the integer workspace is N integer words.

For the KINBAND linear solver, in terms of the problem size N and Jacobian half-bandwidths, the actual size of the real workspace, in `realtype` words, is approximately (2 `mupper`+3 `mlower` +2) N, and the actual size of the integer workspace is N integer words.

`KINDlsGetNumJacEvals`

Call: `flag = KINDlsGetNumJacEvals(kin_mem, &njevals);`

Description: The function `KINDlsGetNumJacEvals` returns the number of calls to the dense Jacobian approximation function.

Arguments: `kin_mem` (`void *`) pointer to the KINSOL memory block.
`njevals` (`long int`) the number of calls to the Jacobian function.

Return value: The return value `flag` (of type `int`) is one of

`KINDLS_SUCCESS`	The optional output value has been successfully set.
`KINDLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINDLS_LMEM_NULL`	The KINDENSE linear solver has not been initialized.

`KINDlsGetNumFuncEvals`

Call: `flag = KINDlsGetNumFuncEvals(kin_mem, &nfevalsLS);`

Description: The function `KINDlsGetNumFuncEvals` returns the number of calls to the user system function used to compute the difference quotient approximation to the dense or banded Jacobian.

Arguments: `kin_mem` (`void *`) pointer to the KINSOL memory block.
`nfevalsLS` (`long int`) the number of calls to the user system function.

Return value: The return value `flag` (of type `int`) is one of

`KINDLS_SUCCESS`	The optional output value has been successfully set.
`KINDLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINDLS_LMEM_NULL`	The KINDENSE or KINBAND linear solver has not been initialized.

Notes: The value `nfevalsLS` is incremented only if the internal difference quotient function is used.

`KINDlsGetLastFlag`

Call: `flag = KINDlsGetLastFlag(kin_mem, &lsflag);`

Description: The function `KINDlsGetLastFlag` returns the last return value from a KINDENSE routine.

Arguments: `kin_mem` (`void *`) pointer to the KINSOL memory block.
`lsflag` (`long int`) the value of the last return flag from a KINDENSE function.

Return value: The return value `flag` (of type `int`) is one of

`KINDLS_SUCCESS`	The optional output value has been successfully set.
`KINDLS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINDLS_LMEM_NULL`	The KINDENSE linear solver has not been initialized.

Notes	If the KINDLS setup function failed (`KINSol` returned `KIN_LSETUP_FAIL`), then `lsflag` is equal to the column index (numbered from one) at which a zero diagonal element was encountered during the LU factorization of the dense Jacobian matrix. For all other failures, `lsflag` is negative.

4.5.5.3 Sparse direct linear solvers optional output functions

The following optional outputs are available from the KINSLS module: number of calls to the Jacobian routine and last return value from a KINSLS function.

`KINSlsGetNumJacEvals`

Call	`flag = KINSlsGetNumJacEvals(kin_mem, &njevals);`
Description	The function `KINSlsGetNumJacEvals` returns the number of calls to the sparse Jacobian approximation function.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `njevals` (`long int`) the number of calls to the Jacobian function.
Return value	The return value `flag` (of type `int`) is one of `KINSLS_SUCCESS` The optional output value has been successfully set. `KINSLS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSLS_LMEM_NULL` The KINSLS linear solver has not been initialized.

`KINSlsGetLastFlag`

Call	`flag = KINSlsGetLastFlag(kin_mem, &lsflag);`
Description	The function `KINSlsGetLastFlag` returns the last return value from a KINSLS routine.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `lsflag` (`long int`) the value of the last return flag from a KINSLS function.
Return value	The return value `flag` (of type `int`) is one of `KINSLS_SUCCESS` The optional output value has been successfully set. `KINSLS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSLS_LMEM_NULL` The KINSLS linear solver has not been initialized.
Notes	

`KINSlsGetReturnFlagName`

Call	`name = KINSlsGetReturnFlagName(lsflag);`
Description	The function `KINSlsGetReturnFlagName` returns the name of the KINSLS constant corresponding to `lsflag`.
Arguments	The only argument, of type `long int`, is a return flag from a KINSLS function.
Return value	The return value is a string containing the name of the corresponding constant.

4.5.5.4 Iterative linear solvers optional output functions

The following optional outputs are available from the KINSPILS modules: workspace requirements, number of linear iterations, number of linear convergence failures, number of calls to the preconditioner setup and solve routines, number of calls to the Jacobian-vector product routine, number of calls to the system function routine for difference quotient Jacobian-vector product approximation, and last return value from a linear solver function.

`KINSpilsGetWorkSpace`

Call	`flag = KINSpilsGetWorkSpace(kin_mem, &lenrwLS, &leniwLS);`
Description	The function `KINSpilsGetWorkSpace` returns the global sizes of the linear solver real and integer workspaces.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `lenrwLS` (`long int`) the number of `realtype` values in the linear solver workspace. `leniwLS` (`long int`) the number of integer values in the linear solver workspace.
Return value	The return value `flag` (of type `int`) is one of: `KINSPILS_SUCCESS` The optional output values have been successfully set. `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSPILS_LMEM_NULL` The linear solver has not been initialized.
Notes	In terms of the problem size N and maximum subspace size `maxl`, the actual size of the real workspace, in `realtype` words, is roughly: (`maxl`+3) $* N+$ `maxl` $*($ `maxl`+4$) + 1$ for KINSPGMR, (2`maxl`+3) $* N+$ `maxl` $*($ `maxl`+4$) + 1$ for KINSPFGMR, $7 * N$ for KINSPBCG, and $11 * N$ for KINSPTFQMR. In a parallel setting, this value is global, summed over all processes.

`KINSpilsGetNumLinIters`

Call	`flag = KINSpilsGetNumLinIters(kin_mem, &nliters);`
Description	The function `KINSpilsGetNumLinIters` returns the cumulative number of linear iterations.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `nliters` (`long int`) the current number of linear iterations.
Return value	The return value `flag` (of type `int`) is one of: `KINSPILS_SUCCESS` The optional output value has been successfully set. `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSPLIS_LMEM_NULL` The linear solver module has not been initialized.

`KINSpilsGetNumConvFails`

Call	`flag = KINSpilsGetNumConvFails(kin_mem, &nlcfails);`
Description	The function `KINSpilsGetNumConvFails` returns the cumulative number of linear convergence failures.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `nlcfails` (`long int`) the current number of linear convergence failures.
Return value	The return value `flag` (of type `int`) is one of: `KINSPILS_SUCCESS` The optional output value has been successfully set. `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSPILS_LMEM_NULL` The linear solver module has not been initialized.

`KINSpilsGetNumPrecEvals`

Call	`flag = KINSpilsGetNumPrecEvals(kin_mem, &npevals);`
Description	The function `KINSpilsGetNumPrecEvals` returns the number of preconditioner evaluations, i.e., the number of calls made to `psetup`.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `npevals` (`long int`) the current number of calls to `psetup`.
Return value	The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The linear solver module has not been initialized.

`KINSpilsGetNumPrecSolves`

Call	`flag = KINSpilsGetNumPrecSolves(kin_mem, &npsolves);`
Description	The function `KINSpilsGetNumPrecSolves` returns the cumulative number of calls made to the preconditioner solve function, `psolve`.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `npsolves` (`long int`) the current number of calls to `psolve`.
Return value	The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The linear solver module has not been initialized.

`KINSpilsGetNumJtimesEvals`

Call	`flag = KINSpilsGetNumJtimesEvals(kin_mem, &njvevals);`
Description	The function `KINSpilsGetNumJtimesEvals` returns the cumulative number made to the Jacobian-vector product function, `jtimes`.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `njvevals` (`long int`) the current number of calls to `jtimes`.
Return value	The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The linear solver module has not been initialized.

`KINSpilsGetNumFuncEvals`

Call	`flag = KINSpilsGetNumFuncEvals(kin_mem, &nfevalsLS);`
Description	The function `KINSpilsGetNumFuncEvals` returns the number of calls to the user system function for difference quotient Jacobian-vector product approximations.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `nfevalsLS` (`long int`) the number of calls to the user system function.
Return value	The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer is `NULL`.
`KINSPILS_LMEM_NULL`	The linear solver module has not been initialized.

Notes	The value `nfevalsLS` is incremented only if the default `KINSpilsDQJtimes` difference quotient function is used.

`KINSpilsGetLastFlag`

Call	`flag = KINSpilsGetLastFlag(kin_mem, &lsflag);`
Description	The function `KINSpilsGetLastFlag` returns the last return value from a KINSPILS routine.
Arguments	`kin_mem` (`void *`) pointer to the KINSOL memory block. `lsflag` (`long int`) the value of the last return flag from a KINSPILS function.
Return value	The return value `flag` (of type `int`) is one of: `KINSPILS_SUCCESS` The optional output value has been successfully set. `KINSPILS_MEM_NULL` The `kin_mem` pointer is `NULL`. `KINSPILS_LMEM_NULL` The linear solver module has not been initialized.
Notes	If the KINSPILS setup function failed (`KINSOL` returned `KIN_LSETUP_FAIL`), `lsflag` will be set to `SPGMR_PSET_FAIL_UNREC`, `SPFGMR_PSET_FAIL_UNREC`, `SPBCG_PSET_FAIL_UNREC`, or `SPTFQMR_PSET_FAIL_UNREC`. If the KINSPGMR solve function failed (`KINSol` returned `KIN_LSOLVE_FAIL`), `lsflag` contains the error return flag from `SpgmrSolve` and will be one of: `SPGMR_MEM_NULL`, indicating that the SPGMR memory is `NULL`; `SPGMR_ATIMES_FAIL_UNREC`, indicating an unrecoverable failure in the Jacobian-times-vector function; `SPGMR_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function, `psolve`, failed unrecoverably; `SPGMR_GS_FAIL`, indicating a failure in the Gram-Schmidt procedure; or `SPGMR_QRSOL_FAIL`, indicating that the matrix R was found to be singular during the QR solve phase. If the KINSPFGMR solve function failed (`KINSol` returned `KIN_LSOLVE_FAIL`), `lsflag` contains the error return flag from `SpfgmrSolve` and will be a similar value to one of the return codes for KINSPGMR. If the KINSPBCG solve function failed (`KINSol` returned `KIN_LSOLVE_FAIL`), `lsflag` contains the error return flag from `SpbcgSolve` and will be one of: `SPBCG_MEM_NULL`, indicating that the SPBCG memory is `NULL`; `SPBCG_ATIMES_FAIL_UNREC`, indicating an unrecoverable failure in the Jacobian-times-vector function; or `SPBCG_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function, `psolve`, failed unrecoverably. If the KINSPTFQMR solve function failed (`KINSol` returned `KIN_LSOLVE_FAIL`), `lsflag` contains the error return flag from `SptfqmrSolve` and will be one of: `SPTFQMR_MEM_NULL`, indicating that the SPTFQMR memory is `NULL`; `SPTFQMR_ATIMES_FAIL_UNREC`, indicating an unrecoverable failure in the $J{*}v$ function; or `SPTFQMR_PSOLVE_FAIL_UNREC`, indicating that the preconditioner solve function, `psolve`, failed unrecoverably.

4.6 User-supplied functions

The user-supplied functions consist of one function defining the nonlinear system, (optionally) a function that handles error and warning messages, (optionally) a function that handles informational messages, (optionally) a function that provides Jacobian-related information for the linear solver, and (optionally) one or two functions that define the preconditioner for use in any of the Krylov iterative algorithms.

4.6.1 Problem-defining function

The user must provide a function of type `KINSysFn` defined as follows:

`KINSysFn`

Definition	`typedef int (*KINSysFn)(N_Vector u, N_Vector fval, void *user_data);`

Purpose	This function computes $F(u)$ (or $G(u)$ for fixed-point iteration and Anderson acceleration) for a given value of the vector u.
Arguments	`u` is the current value of the variable vector, u. `fval` is the output vector $F(u)$. `user_data` is a pointer to user data, the pointer `user_data` passed to `KINSetUserData`.
Return value	A `KINSysFn` function should return 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if it failed unrecoverably (in which case the solution process is halted and `KIN_SYSFUNC_FAIL` is returned).
Notes	Allocation of memory for `fval` is handled within KINSOL.

4.6.2 Error message handler function

As an alternative to the default behavior of directing error and warning messages to the file pointed to by `errfp` (see `KINSetErrFile`), the user may provide a function of type `KINErrHandlerFn` to process any such messages. The function type `KINErrHandlerFn` is defined as follows:

`KINErrHandlerFn`

Definition
```
typedef void (*KINErrHandlerFn)(int error_code, const char *module,
                                const char *function, char *msg,
                                void *eh_data);
```

Purpose This function processes error and warning messages from KINSOL and its sub-modules.

Arguments	
`error_code`	is the error code.
`module`	is the name of the KINSOL module reporting the error.
`function`	is the name of the function in which the error occurred.
`msg`	is the error message.
`eh_data`	is a pointer to user data, the same as the `eh_data` parameter passed to `KINSetErrHandlerFn`.

Return value A `KINErrHandlerFn` function has no return value.

Notes `error_code` is negative for errors and positive (`KIN_WARNING`) for warnings. If a function that returns a pointer to memory encounters an error, it sets `error_code` to 0.

4.6.3 Informational message handler function

As an alternative to the default behavior of directing informational (meaning non-error) messages to the file pointed to by `infofp` (see `KINSetInfoFile`), the user may provide a function of type `KINInfoHandlerFn` to process any such messages. The function type `KINInfoHandlerFn` is defined as follows:

`KINInfoHandlerFn`

Definition
```
typedef void (*KINInfoHandlerFn)(const char *module, const char *function,
                                 char *msg, void *ih_data);
```

Purpose This function processes informational messages from KINSOL and its sub-modules.

Arguments	
`module`	is the name of the KINSOL module reporting the information.
`function`	is the name of the function reporting the information.
`msg`	is the message.
`ih_data`	is a pointer to user data, the same as the `ih_data` parameter passed to `KINSetInfoHandlerFn`.

Return value A `KINInfoHandlerFn` function has no return value.

4.6.4 Jacobian information (direct method with dense Jacobian)

If the direct linear solver with dense treatment of the Jacobian is used (`KINDense` or `KINLapackDense` is called in Step 7 of §4.4), the user may provide a function of type `KINDlsDenseJacFn` defined by

`KINDlsDenseJacFn`

Definition
```
typedef int (*KINDlsDenseJacFn)(long int N, N_Vector u, N_Vector fu,
                                DlsMat J, void *user_data,
                                N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the dense Jacobian $J(u)$ or an approximation to it.

Arguments

`N`	is the problem size.
`u`	is the current (unscaled) iterate.
`fu`	is the current value of the vector $F(u)$.
`J`	is the output approximate Jacobian matrix, $J = \partial F/\partial u$.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.
`tmp1`	
`tmp2`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `KINDenseJacFn` as temporary storage or work space.

Return value A function of type `KINDlsDenseJacFn` should return 0 if successful or a non-zero value otherwise.

Notes A user-supplied dense Jacobian function must load the `N` by `N` dense matrix `J` with an approximation to the Jacobian matrix $J(u)$ at `u`. Only nonzero elements need to be loaded into `J` because `J` is set to the zero matrix before the call to the Jacobian function. The type of `J` is `DlsMat`.

The accessor macros `DENSE_ELEM` and `DENSE_COL` allow the user to read and write dense matrix elements without making explicit references to the underlying representation of the `DlsMat` type. `DENSE_ELEM(J, i, j)` references the (`i`, `j`)-th element of the dense matrix `J` (`i`, `j`$= 0 \ldots N-1$). This macro is for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices m and n running from 1 to N, the Jacobian element $J_{m,n}$ can be loaded with the statement `DENSE_ELEM(J, m-1, n-1) =` $J_{m,n}$. Alternatively, `DENSE_COL(J, j)` returns a pointer to the storage for the `j`th column of `J` (`j`$= 0 \ldots N-1$), and the elements of the `j`th column are then accessed via ordinary array indexing. Thus $J_{m,n}$ can be loaded with the statements `col_n = DENSE_COL(J, n-1); col_n[m-1] =` $J_{m,n}$. For large problems, it is more efficient to use `DENSE_COL` than to use `DENSE_ELEM`. Note that both of these macros number rows and columns starting from 0, not 1.

The `DlsMat` type and the accessor macros `DENSE_ELEM` and `DENSE_COL` are documented in §8.1.3.

If the user's `KINDlsDenseJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These might include the scale vectors and the unit roundoff. To obtain the scale vectors, the user will need to add to `user_data` pointers to `u_scale` and/or `f_scale` as needed. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

For the sake of uniformity, the argument `N` is of type `long int`, even in the case that the Lapack dense solver is to be used.

4.6.5 Jacobian information (direct method with banded Jacobian)

If the direct linear solver with banded treatment of the Jacobian is used (`KINBand` or `KINLapackBand` is called in Step 7 of §4.4), the user may provide a function of type `KINDlsBandJacFn` defined by:

`KINDlsBandJacFn`

Definition
```
typedef int (*KINDlsBandJacFn)(long int N, long int mupper,
                               long int mlower, N_Vector u, N_Vector fu,
                               DlsMat J, void *user_data,
                               N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the banded Jacobian $J(u)$ or a banded approximation to it.

Arguments

`N`	is the problem size.
`mlower`	
`mupper`	are the lower and upper half-bandwidths of the Jacobian.
`u`	is the current (unscaled) iterate.
`fu`	is the current value of the vector $F(u)$.
`J`	is the output approximate Jacobian matrix, $J = \partial F/\partial u$.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.
`tmp1`	
`tmp2`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `KINBandJacFn` as temporary storage or work space.

Return value A function of type `KINDlsBandJacFn` should return 0 if successful or a non-zero value otherwise.

Notes A user-supplied band Jacobian function must load the band matrix `J` of type `DlsMat` with the elements of the Jacobian $J(u)$ at `u`. Only nonzero elements need to be loaded into `J` because `J` is preset to zero before the call to the Jacobian function.

The accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` allow the user to read and write band matrix elements without making specific references to the underlying representation of the `DlsMat` type. `BAND_ELEM(J, i, j)` references the (`i`, `j`)th element of the band matrix `J`, counting from 0. This macro is for use in small problems in which efficiency of access is not a major concern. Thus, in terms of indices m and n running from 1 to N with (m, n) within the band defined by `mupper` and `mlower`, the Jacobian element $J_{m,n}$ can be loaded with the statement `BAND_ELEM(J, m-1, n-1)` `=` $J_{m,n}$. The elements within the band are those with `-mupper` $\leq$ `m-n` $\leq$ `mlower`. Alternatively, `BAND_COL(J, j)` returns a pointer to the diagonal element of the `j`th column of `J`, and if we assign this address to `realtype *col_j`, then the `i`th element of the `j`th column is given by `BAND_COL_ELEM(col_j, i, j)`, counting from 0. Thus for (m, n) within the band, $J_{m,n}$ can be loaded by setting `col_n = BAND_COL(J, n-1);` `BAND_COL_ELEM(col_n, m-1, n-1) =` $J_{m,n}$. The elements of the `j`th column can also be accessed via ordinary array indexing, but this approach requires knowledge of the underlying storage for a band matrix of type `DlsMat`. The array `col_n` can be indexed from $-$`mupper` to `mlower`. For large problems, it is more efficient to use the combination of `BAND_COL` and `BAND_COL_ELEM` than to use the `BAND_ELEM`. As in the dense case, these macros all number rows and columns starting from 0, not 1.

The `DlsMat` type and the accessor macros `BAND_ELEM`, `BAND_COL`, and `BAND_COL_ELEM` are documented in §8.1.4.

If the user's `KINDlsBandJacFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These might include the scale vectors and the unit roundoff. To obtain the scale vectors, the user will need to add to `user_data` pointers to `u_scale` and/or `f_scale` as needed. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

For the sake of uniformity, the arguments `N`, `mlower`, and `mupper` are of type `long int`, even in the case that the Lapack band solver is to be used.

4.6.6 Jacobian information (direct method with sparse Jacobian)

If the direct linear solver with sparse treatment of the Jacobian is used (`KINKLU` or `KINSuperLUMT` is called in Step 7 of §4.4), the user may provide a function of type `KINSlsSparseJacFn` defined by

`KINSlsSparseJacFn`

Definition
```
typedef int (*KINSlsSparseJacFn)(N_Vector u, N_Vector fu,
                                 SlsMat J, void *user_data,
                                 N_Vector tmp1, N_Vector tmp2);
```

Purpose This function computes the sparse Jacobian $J(u)$ or an approximation to it.

Arguments
- `u` is the current (unscaled) iterate.
- `fu` is the current value of the vector $F(u)$.
- `J` is the output approximate Jacobian matrix, $J = \partial F/\partial u$.
- `user_data` is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.
- `tmp1`
- `tmp2` are pointers to memory allocated for variables of type `N_Vector` which can be used by `KINSlsSparseJacFn` as temporary storage or work space.

Return value A function of type `KINSlsSparseJacFn` should return 0 if successful or a non-zero value otherwise.

Notes A user-supplied sparse Jacobian function must load the compressed-sparse-column matrix `J` with an approximation to the Jacobian matrix $J(u)$ at the point (`u`). Storage for `J` already exists on entry to this function, although the user should ensure that sufficient space is allocated in `J` to hold the nonzero values to be set; if the existing space is insufficient the user may reallocate the data and row index arrays as needed. The type of `J` is `SlsMat`, and the amount of allocated space is available within the `SlsMat` structure as `NNZ`. The `SlsMat` type is further documented in the Section §8.2.

If the user's `KINSlsSparseJacFn` function uses difference quotient approximations to set the specific nonzero matrix entries, then it may need to access quantities not in the argument list. These might include the scale vectors and the unit roundoff. To obtain the scale vectors, the user will need to add to `user_data` pointers to `u_scale` and/or `f_scale` as needed. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.6.7 Jacobian information (matrix-vector product)

If one of the Krylov iterative linear solvers SPGMR, SPBCG, or SPTFQMR is selected (`KINSp*` is called in step 7 of §4.4), the user may provide a `jtimes` function of type `KINSpilsJacTimesVecFn` to compute products Jv. If such a function is not supplied, the default is a difference quotient approximation of these products.

`KINSpilsJacTimesVecFn`

Definition
```
typedef int (*KINSpilsJacTimesVecFn)(N_Vector v, N_Vector Jv,
                                     N_Vector u, booleantype new_u,
                                     void *user_data);
```

Purpose This `jtimes` function computes the product Jv (or an approximation to it).

Arguments
- `v` is the vector by which the Jacobian must be multiplied to the right.
- `Jv` is the computed output vector.
- `u` is the current value of the dependent variable vector.

`new_u`	is a flag, input from KINSOL and possibly reset by the user's `jtimes` function, indicating whether the iterate vector `u` has been updated since the last call to `jtimes`. This is useful if the `jtimes` function computes and saves Jacobian data that depends on `u` for use in computing $J(u)v$. The input value of `new_u` is `TRUE` following an update by KINSOL, and in that case any saved Jacobian data depending on `u` should be recomputed. The `jtimes` routine should then set `new_u` to `FALSE`, so that on subsequent calls to `jtimes` with the same `u`, the saved data can be reused.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.

Return value The value to be returned by the Jacobian-times-vector function should be 0 if successful. If a recoverable failure occurred, the return value should be positive. In this case, KINSOL will attempt to correct by calling the preconditioner setup function. If this information is current, KINSOL halts. If the Jacobian-times-vector function encounters an unrecoverable error, it should return a negative value, prompting KINSOL to halt.

Notes If a user-defined routine is not given, then an internal `jtimes` function, using a difference quotient approximation, is used.

If the user's `KINSpilsJacTimesVecFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These might include the scale vectors and the unit roundoff. To obtain the scale vectors, the user will need to add to `user_data` pointers to `u_scale` and/or `f_scale` as needed. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

4.6.8 Preconditioning (linear system solution)

If preconditioning is used, then the user must provide a C function to solve the linear system $Pz = r$ where P is the preconditioner matrix, approximating (at least crudely) the system Jacobian $J = \partial F/\partial u$. This function must be of type `KINSpilsPrecSolveFn`, defined as follows:

`KINSpilsPrecSolveFn`

Definition
```
typedef int (*KINSpilsPrecSolveFn)(N_Vector u, N_Vector uscale,
                                   N_Vector fval, N_Vector fscale,
                                   N_Vector v, void *user_data,
                                   N_Vector tmp);
```

Purpose This function solves the preconditioning system $Pz = r$.

Arguments

`u`	is the current (unscaled) value of the iterate.
`uscale`	is a vector containing diagonal elements of the scaling matrix for `u`.
`fval`	is the vector $F(u)$ evaluated at `u`.
`fscale`	is a vector containing diagonal elements of the scaling matrix for `fval`.
`v`	on input, `v` is set to the right-hand side vector of the linear system, `r`. On output, `v` must contain the solution `z` of the linear system $Pz = r$.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to the function `KINSetUserData`.
`tmp`	is a pointer to memory allocated for a variable of type `N_Vector` which can be used for work space.

Return value The value to be returned by the preconditioner solve function is a flag indicating whether it was successful. This value should be 0 if successful, positive for a recoverable error, and negative for an unrecoverable error.

Notes If the preconditioner solve function fails recoverably and if the preconditioner information (set by the preconditioner setup function) is out of date, KINSOL attempts to correct by calling the setup function. If the preconditioner data is current, KINSOL halts.

4.6.9 Preconditioning (Jacobian data)

If the user's preconditioner requires that any Jacobian-related data be evaluated or preprocessed, then this needs to be done in a user-supplied C function of type `KINSpilsPrecSetupFn`, defined as follows:

`KINSpilsPrecSetupFn`

Definition
```
typedef int (*KINSpilsPrecSetupFn)(N_Vector u, N_Vector uscale,
                                   N_Vector fval, N_Vector fscale,
                                   void *user_data, N_Vector tmp1,
                                   N_Vector tmp2);
```

Purpose: This function evaluates and/or preprocesses Jacobian-related data needed by the preconditioner solve function.

Arguments: The arguments of a `KINSpilsPrecSetupFn` are as follows:

`u`	is the current (unscaled) value of the iterate.
`uscale`	is a vector containing diagonal elements of the scaling matrix for `u`.
`fval`	is the vector $F(u)$ evaluated at `u`.
`fscale`	is a vector containing diagonal elements of the scaling matrix for `fval`.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to the function `KINSetUserData`.
`tmp1`	
`tmp2`	are pointers to memory allocated for variables of type `N_Vector` which can be used by `KINSpilsPrecSetupFn` as temporary storage or work space.

Return value: The value to be returned by the preconditioner setup function is a flag indicating whether it was successful. This value should be 0 if successful, any other value resulting in halting the KINSOL solver.

Notes: The user-supplied preconditioner setup subroutine should compute the right preconditioner matrix P (stored in the memory block referenced by the `user_data` pointer) used to form the scaled preconditioned linear system

$$(D_F J(u) P^{-1} D_u^{-1}) \cdot (D_u P x) = -D_F F(u)\,,$$

where D_u and D_F denote the diagonal scaling matrices whose diagonal elements are stored in the vectors `uscale` and `fscale`, respectively.

The preconditioner setup routine will not be called prior to every call made to the preconditioner solve function, but will instead be called only as often as necessary to achieve convergence of the Newton iteration.

If the user's `KINSpilsPrecSetupFn` function uses difference quotient approximations, it may need to access quantities not in the call list. These might include the scale vectors and the unit roundoff. To obtain the scale vectors, the user will need to add to `user_data` pointers to `u_scale` and/or `f_scale` as needed. The unit roundoff can be accessed as `UNIT_ROUNDOFF` defined in `sundials_types.h`.

If the preconditioner solve routine requires no preparation, then a preconditioner setup function need not be given.

4.7 A parallel band-block-diagonal preconditioner module

The efficiency of Krylov iterative methods for the solution of linear systems can be greatly enhanced through preconditioning. For problems in which the user cannot define a more effective, problem-specific preconditioner, KINSOL provides a band-block-diagonal preconditioner module KINBBDPRE, to be used with the parallel `N_Vector` module described in §6.2.

This module provides a preconditioner matrix for KINSOL that is block-diagonal with banded blocks. The blocking corresponds to the distribution of the dependent variable vector u amongst the processes. Each preconditioner block is generated from the Jacobian of the local part (associated with the current process) of a given function $G(u)$ approximating $F(u)$ ($G = F$ is allowed). The blocks are generated by each process via a difference quotient scheme, utilizing a specified band structure. This structure is given by upper and lower half-bandwidths, `mudq` and `mldq`, defined as the number of non-zero diagonals above and below the main diagonal, respectively. However, from the resulting approximate Jacobain blocks, only a matrix of bandwidth `mukeep` + `mlkeep` +1 is retained.

Neither pair of parameters need be the true half-bandwidths of the Jacobian of the local block of G, if smaller values provide a more efficient preconditioner. Such an efficiency gain may occur if the couplings in the system outside a certain bandwidth are considerably weaker than those within the band. Reducing `mukeep` and `mlkeep` while keeping `mudq` and `mldq` at their true values, discards the elements outside the narrower band. Reducing both pairs has the additional effect of lumping the outer Jacobian elements into the computed elements within the band, and requires more caution and experimentation to see whether the lower cost of narrower band matrices offsets the loss of accuracy in the blocks.

The KINBBDPRE module calls two user-provided functions to construct P: a required function `Gloc` (of type `KINLocalFn`) which approximates the nonlinear system function function $G(u) \approx F(u)$ and which is computed locally, and an optional function `Gcomm` (of type `KINCommFn`) which performs all interprocess communication necessary to evaluate the approximate function G. These are in addition to the user-supplied nonlinear system function that evaluates $F(u)$. Both functions take as input the same pointer `user_data` as that passed by the user to `KINSetUserData` and passed to the user's function `func`, and neither function has a return value. The user is responsible for providing space (presumably within `user_data`) for components of `u` that are communicated by `Gcomm` from the other processes, and that are then used by `Gloc`, which should not do any communication.

`KINLocalFn`

Definition
```
typedef void (*KINLocalFn)(long int Nlocal, N_Vector u,
                           N_Vector gval, void *user_data);
```

Purpose: This `Gloc` function computes $G(\mathtt{u})$, and outputs the resulting vector as `gval`.

Arguments:

`Nlocal`	is the local vector length.
`u`	is the current value of the iterate.
`gval`	is the output vector.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.

Return value: A `KINLocalFn` function type does not have a return value.

Notes: This function must assume that all interprocess communication of data needed to calculate `gval` has already been done, and this data is accessible within `user_data`.

Memory for `u` and `gval` is handled within the preconditioner module.

The case where G is mathematically identical to F is allowed.

`KINCommFn`

Definition
```
typedef void (*KINCommFn)(long int Nlocal, N_Vector u, void *user_data);
```

Purpose: This `Gcomm` function performs all interprocess communications necessary for the execution of the `Gloc` function above, using the input vector `u`.

Arguments:

`Nlocal`	is the local vector length.
`u`	is the current value of the iterate.
`user_data`	is a pointer to user data, the same as the `user_data` parameter passed to `KINSetUserData`.

Return value A KINCommFn function type does not have a return value.

Notes The Gcomm function is expected to save communicated data in space defined within the structure user_data.

Each call to the Gcomm function is preceded by a call to the system function func with the same u argument. Thus Gcomm can omit any communications done by func if relevant to the evaluation of Gloc. If all necessary communication was done in func, then Gcomm = NULL can be passed in the call to KINBBDPrecInit (see below).

Besides the header files required for the solution of a nonlinear problem (see §4.3), to use the KINBBDPRE module, the main program must include the header file kinbbdpre.h which declares the needed function prototypes.

The following is a summary of the usage of this module and describes the sequence of calls in the user main program. Steps that are unchanged from the user main program presented in §4.4 are grayed out.

1. **Initialize MPI**
2. **Set problem dimensions**
3. **Set vector with initial guess**
4. **Create KINSOL object**
5. **Set optional inputs**
6. **Allocate internal memory**
7. **Attach iterative linear solver, one of:**

   ```
   flag = KINSpgmr(kin_mem, maxl);
   flag = KINSpfgmr(kin_mem, maxl);
   flag = KINSpbcg(kin_mem, maxl);
   flag = KINSptfqmr(kin_mem, maxl);
   ```

8. **Initialize the KINBBDPRE preconditioner module**

   Specify the upper and lower half-bandwidth pairs (mudq, mldq) and (mukeep, mlkeep), and call

   ```
   flag = KINBBDPrecInit(kin_mem, Nlocal, mudq, mldq,
                         mukeep, mlkeep, dq_rel_u, Gloc, Gcomm);
   ```

   to allocate memory for and initialize the internal preconditoner data. The last two arguments of KINBBDPrecInit are the two user-supplied functions described above.

9. **Set linear solver optional inputs**

   Note that the user should not overwrite the preconditioner data, setup function, or solve function through calls to KINSPILS optional input functions.

10. **Solve problem**
11. **Get optional output**

    Additional optional outputs associated with KINBBDPRE are available by way of two routines described below, KINBBDPrecGetWorkSpace and KINBBDPrecGetNumGfnEvals.

12. **Deallocate memory for solution vector**
13. **Free solver memory**

14. **Finalize MPI**

The user-callable function that initializes KINBBDPRE (step 8), is described in more detail below.

`KINBBDPrecInit`

Call
```
flag = KINBBDPrecInit(kin_mem, Nlocal, mudq, mldq,
                      mukeep, mlkeep, dq_rel_u, Gloc, Gcomm);
```

Description The function `KINBBDPrecInit` initializes and allocates memory for the KINBBDPRE preconditioner.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`Nlocal`	(`long int`) local vector length.
`mudq`	(`long int`) upper half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mldq`	(`long int`) lower half-bandwidth to be used in the difference-quotient Jacobian approximation.
`mukeep`	(`long int`) upper half-bandwidth of the retained banded approximate Jacobian block.
`mlkeep`	(`long int`) lower half-bandwidth of the retained banded approximate Jacobian block.
`dq_rel_u`	(`realtype`) the relative increment in components of `u` used in the difference quotient approximations. The default is `dq_rel_u`= $\sqrt{\text{unit roundoff}}$, which can be specified by passing `dq_rel_u`= 0.0.
`Gloc`	(`KINLocalFn`) the C function which computes the approximation $G(u) \approx F(u)$.
`Gcomm`	(`KINCommFn`) the optional C function which performs all interprocess communication required for the computation of $G(u)$.

Return value The return value `flag` (of type `int`) is one of

`KINSPILS_SUCCESS`	The call to `KINBBDPrecInit` was successful.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer was `NULL`.
`KINSPILS_MEM_FAIL`	A memory allocation request has failed.
`KINSPILS_LMEM_NULL`	A KINSPILS linear solver was not attached.
`KINSPILS_ILL_INPUT`	The supplied vector implementation was not compatible with block band preconditioner.

Notes If one of the half-bandwidths `mudq` or `mldq` to be used in the difference-quotient calculation of the approximate Jacobian is negative or exceeds the value `Nlocal`−1, it is replaced with 0 or `Nlocal`−1 accordingly.

The half-bandwidths `mudq` and `mldq` need not be the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide greater efficiency.

Also, the half-bandwidths `mukeep` and `mlkeep` of the retained banded approximate Jacobian block may be even smaller, to reduce storage and computation costs further.

For all four half-bandwidths, the values need not be the same for every process.

The following two optional output functions are available for use with the KINBBDPRE module:

`KINBBDPrecGetWorkSpace`

Call
```
flag = KINBBDPrecGetWorkSpace(kin_mem, &lenrwBBDP, &leniwBBDP);
```

Description The function `KINBBDPrecGetWorkSpace` returns the local KINBBDPRE real and integer workspace sizes.

Arguments `kin_mem` (`void *`) pointer to the KINSOL memory block.

`lenrwBBDP`	(`long int`) local number of `realtype` values in the KINBBDPRE workspace.
`leniwBBDP`	(`long int`) local number of integer values in the KINBBDPRE workspace.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer was `NULL`.
`KINSPILS_PMEM_NULL`	The KINBBDPRE preconditioner has not been initialized.

Notes In terms of the local vector dimension `Nlocal` and `smu` $=\min(N_l-1,$ `mukeep` $+$ `mlkeep`$)$, the actual size of the real workspace is $(2$ `mlkeep` $+$ `mukeep` $+$ `smu` $+2)$ `Nlocal` `realtype` words, and the actual size of the integer workspace is `Nlocal` integer words. These values are local to the current processor.

The workspaces referred to here exist in addition to those given by the corresponding `KINSp*GetWorkSpace` function.

`KINBBDPrecGetNumGfnEvals`

Call `flag = KINBBDPrecGetNumGfnEvals(kin_mem, &ngevalsBBDP);`

Description The function `KINBBDPrecGetNumGfnEvals` returns the number of calls to the user `Gloc` function due to the difference quotient approximation of the Jacobian blocks used within KINBBDPRE's preconditioner setup function.

Arguments

`kin_mem`	(`void *`) pointer to the KINSOL memory block.
`ngevalsBBDP`	(`long int`) the number of calls to the user `Gloc` function.

Return value The return value `flag` (of type `int`) is one of:

`KINSPILS_SUCCESS`	The optional output value has been successfully set.
`KINSPILS_MEM_NULL`	The `kin_mem` pointer was `NULL`.
`KINSPILS_PMEM_NULL`	The KINBBDPRE preconditioner has not been initialized.

In addition to the `ngevalsBBDP` `Gloc` evaluations, the costs associated with KINBBDPRE also include `nlinsetups` LU factorizations, `nlinsetups` calls to `Gcomm`, `npsolves` banded backsolve calls, and `nfevalsLS` right-hand side function evaluations, where `nlinsetups` is an optional KINSOL output and `npsolves` and `nfevalsLS` are linear solver optional outputs (see §4.5.5).

Chapter 5

FKINSOL, an Interface Module for FORTRAN Applications

The FKINSOL interface module is a package of C functions which support the use of the KINSOL solver, for the solution of nonlinear systems $F(u) = 0$, in a mixed FORTRAN/C setting. While KINSOL is written in C, it is assumed here that the user's calling program and user-supplied problem-defining routines are written in FORTRAN. This package provides the necessary interface to KINSOL for all supplied serial and parallel NVECTOR implementations.

5.1 Important note on portability

In this package, the names of the interface functions, and the names of the FORTRAN user routines called by them, appear as dummy names which are mapped to actual values by a series of definitions in the header files. By default, those mapping definitions depend in turn on the C macro `F77_FUNC` defined in the header file `sundials_config.h`. The mapping defined by `F77_FUNC` in turn transforms the C interface names to match the name-mangling approach used by the supplied Fortran compiler.

By "name-mangling", we mean that due to the case-independent nature of the FORTRAN language, FORTRAN compilers convert all subroutine and object names to use either all lower-case or all upper-case characters, and append either zero, one or two underscores as a prefix or suffix to the name. For example, the FORTRAN subroutine `MyFunction()` will be changed to one of `myfunction`, `MYFUNCTION`, `myfunction__`, `MYFUNCTION_`, and so on, depending on the FORTRAN compiler used.

SUNDIALS determines this name-mangling scheme at configuration time (see Appendix A).

5.2 Fortran Data Types

Throughout this documentation, we will refer to data types according to their usage in C. The equivalent types to these may vary, depending on your computer architecture and on how SUNDIALS was compiled (see Appendix A). A FORTRAN user should first determine the equivalent types for their architecture and compiler, and then take care that all arguments passed through this FORTRAN/C interface are declared of the appropriate type.

Integers: SUNDIALS uses both `int` and `long int` types:

- `int` – equivalent to an `INTEGER` or `INTEGER*4` in FORTRAN
- `long int` – this will depend on the computer architecture:
  - 32-bit architecture – equivalent to an `INTEGER` or `INTEGER*4` in FORTRAN
  - 64-bit architecture – equivalent to an `INTEGER*8` in FORTRAN

Real numbers: As discussed in Appendix A, at compilation SUNDIALS allows the configuration option `--with-precision`, that accepts values of `single`, `double` or `extended` (the default is `double`). This choice dictates the size of a `realtype` variable. The corresponding FORTRAN types for these `realtype` sizes are:

- `single` – equivalent to a `REAL` or `REAL*4` in FORTRAN
- `double` – equivalent to a `DOUBLE PRECISION` or `REAL*8` in FORTRAN
- `extended` – equivalent to a `REAL*16` in FORTRAN

5.3 FKINSOL routines

The user-callable functions, with the corresponding KINSOL functions, are as follows:

- Interface to the NVECTOR modules
  - `FNVINITS` (defined by NVECTOR_SERIAL) interfaces to `N_VNewEmpty_Serial`.
  - `FNVINITOMP` (defined by NVECTOR_OPENMP) interfaces to `N_VNewEmpty_OpenMP`.
  - `FNVINITPTS` (defined by NVECTOR_PTHREADS) interfaces to `N_VNewEmpty_Pthreads`.
  - `FNVINITP` (defined by NVECTOR_PARALLEL) interfaces to `N_VNewEmpty_Parallel`.
- Interface to the main KINSOL module
  - `FKINMALLOC` interfaces to `KINCreate`, `KINSetUserData`, and `KINInit`.
  - `FKINSETIIN` and `FKINSETRIN` interface to `KINSet*` functions.
  - `FKINSETVIN` interfaces to `KINSetConstraints`.
  - `FKINSOL` interfaces to `KINSol`, `KINGet*` functions, and to the optional output functions for the selected linear solver module.
  - `FKINFREE` interfaces to `KINFree`.
- Interface to the linear solver modules
  - `FKINDENSE` interfaces to `KINDense`.
  - `FKINDENSESETJAC` interfaces to `KINDlsSetDenseJacFn`.
  - `FKINLAPACKDENSE` interfaces to `KINLapackDense`.
  - `FKINLAPACKDENSESETJAC` interfaces to `KINDlsSetDenseJacFn`.
  - `FKINBAND` interfaces to `KINBand`.
  - `FKINBANDSETJAC` interfaces to `KINDlsSetBandJacFn`.
  - `FKINLAPACKBAND` interfaces to `KINLapackBand`.
  - `FKINLAPACKBANDSETJAC` interfaces to `KINDlsSetBandJacFn`.
  - `FKINKLU` interfaces to `KINKLU`.
  - `FKINKLUREINIT` interfaces to `KINKLUReInit`.
  - `FKINSUPERLUMT` interfaces to `KINSuperLUMT`.
  - `FKINSPGMR` interfaces to `KINSpgmr` and SPGMR optional input functions.
  - `FKINSPFGMR` interfaces to `KINSpfgmr` and SPFGMR optional input functions.
  - `FKINSPBCG` interfaces to `KINSpbcg` and SPBCG optional input functions.
  - `FKINSPTFQMR` interfaces to `KINSptfqmr` and SPTFQMR optional input functions.
  - `FKINSPILSSETJAC` interfaces to `KINSpilsSetJacTimesVecFn`.
  - `FKINSPILSSETPREC` interfaces to `KINSpilsSetPreconditioner`.

The user-supplied functions, each listed with the corresponding internal interface function which calls it (and its type within KINSOL), are as follows:

FKINSOL routine (FORTRAN, user-supplied)	KINSOL function (C, interface)	KINSOL type of interface function
`FKFUN`	`FKINfunc`	`KINSysFn`
`FKDJAC`	`FKINDenseJac`	`KINDlsDenseJacFn`
	`FKINLapackDenseJac`	`KINDlsDenseJacFn`
`FKBJAC`	`FKINBandJac`	`KINDlsBandJacFn`
	`FKINLapackBandJac`	`KINDlsBandJacFn`
`FKINSPJAC`	`FKINSparseJac`	`KINSlsSparseJacFn`
`FKPSET`	`FKINPSet`	`KINSpilsPrecSetupFn`
`FKPSOL`	`FKINPSol`	`KINSpilsPrecSolveFn`
`FKJTIMES`	`FKINJtimes`	`KINSpilsJacTimesVecFn`

In contrast to the case of direct use of KINSOL, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program.

5.4 Usage of the FKINSOL interface module

The usage of FKINSOL requires calls to a few different interface functions, depending on the method options selected, and one or more user-supplied routines which define the problem to be solved. These function calls and user routines are summarized separately below. Some details are omitted, and the user is referred to the description of the corresponding KINSOL functions for information on the arguments of any given user-callable interface routine, or of a given user-supplied function called by an interface function.

In the instructions below, steps marked [**S**] apply to the NVECTOR module NVECTOR_SERIAL, steps marked [**O**] apply to NVECTOR_OPENMP, steps marked [**T**] apply to NVECTOR_PTHREADS, while steps marked [**P**] apply to NVECTOR_PARALLEL,

1. **Nonlinear system function specification**

   The user must, in all cases, supply the following FORTRAN routine

   ```
   SUBROUTINE FKFUN (U, FVAL, IER)
   DIMENSION U(*), FVAL(*)
   ```

   It must set the `FVAL` array to $F(u)$, the system function, as a function of `U` $= u$. `IER` is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if it failed unrecoverably (in which case the solution process is halted).

2. NVECTOR **module initialization**

   [**S**] To initialize the serial NVECTOR module, the user must make the following call:

   ```
   CALL FNVINITS (KEY, NEQ, IER)
   ```

   where `KEY` is the solver id (`KEY` = 3 for KINSOL), `NEQ` is the size of vectors, and `IER` is a return completion flag which is 0 on success and -1 if a failure occurred.

   [**O**] To initialize the NVECTOR_OPENMP NVECTOR module, the user must make the following call:

   ```
   CALL FNVINITOMP(KEY, NEQ, NUMTHREADS, IER)
   ```

where KEY is the solver id (KEY = 3 for KINSOL), NEQ is the size of vectors, NUMTHREADS is the number of threads, and IER is a return completion flag which is 0 on success and −1 if a failure occurred.

[**T**] To initialize the NVECTOR_PTHREADS NVECTOR module, the user must make the following call:

```
CALL FNVINITPTS(KEY, NEQ, NUMTHREADS, IER)
```

where KEY is the solver id (KEY = 3 for KINSOL), NEQ is the size of vectors, NUMTHREADS is the number of threads, and IER is a return completion flag which is 0 on success and −1 if a failure occurred.

[**P**] To initialize the MPI-based distributed memory parallel vector module, the user must make the following call:

```
CALL FNVINITP (COMM, KEY, NLOCAL, NGLOBAL, IER)
```

in which the arguments are: COMM = MPI communicator, KEY = 3 for KINSOL, NLOCAL = the local size of vectors on this processor, and NGLOBAL = the system size (and the global size of all vectors, equal to the sum of all values of NLOCAL). The return flag IER is set to 0 on a successful return and to −1 otherwise.

NOTE: The integers NEQ, NLOCAL, and NGLOBAL should be declared so as to match C type long int.

! If the header file sundials_config.h defines SUNDIALS_MPI_COMM_F2C to be 1 (meaning the MPI implementation used to build SUNDIALS includes the MPI_Comm_f2c function), then COMM can be any valid MPI communicator. Otherwise, MPI_COMM_WORLD will be used, so just pass an integer value as a placeholder.

3. **Problem specification**

To set various problem and solution parameters and allocate internal memory, make the following call:

FKINMALLOC

Call	CALL FKINMALLOC (IOUT, ROUT, IER)
Description	This function specifies the optional output arrays, allocates internal memory, and initializes KINSOL.
Arguments	IOUT is an integer array for integer optional outputs. ROUT is a real array for real optional outputs.
Return value	IER is the return completion flag. Values are 0 for successful return and −1 otherwise. See printed message for details in case of failure.
Notes	The user integer data array IOUT must be declared as INTEGER*4 or INTEGER*8 according to the C type long int. The optional outputs associated with the main KINSOL integrator are listed in Table 5.2.

4. **Set optional inputs**

Call FKINSETIIN, FKINSETRIN, and/or FKINSETVIN, to set desired optional inputs, if any. See §5.5 for details.

5. **Linear solver specification**

The solution method in KINSOL involves the solution of linear systems related to the Jacobian of the nonlinear system. The user of FKINSOL must call a routine with a specific name to make the desired choice of linear solver.

[S] Dense treatment of the linear system

To use the direct dense linear solver based on the internal KINSOL implementation, the user must make the call:

```
CALL FKINDENSE (NEQ, IER)
```

where `NEQ` is the size of the nonlinear system. The argument `IER` is an error return flag which is 0 for success , -1 if a memory allocation failure occurred, or -2 for illegal input.

Alternatively, to use the Lapack-based direct dense linear solver, the user must make the call:

```
CALL FKINLAPACKDENSE(NEQ, IER)
```

where the arguments have the same meanings as for `FKINDENSE`, except that here `NEQ` must be declared so as to match C type `int`.

As an option when using the DENSE linear solver, the user may supply a routine that computes a dense approximation of the system Jacobian $J = \partial F/\partial u$. If supplied, it must have the following form:

```
SUBROUTINE FKDJAC (NEQ, U, FVAL, DJAC, WK1, WK2, IER)
DIMENSION U(*), FVAL(*), DJAC(NEQ,*), WK1(*), WK2(*)
```

Typically this routine will use only `NEQ`, `U`, and `DJAC`. It must compute the Jacobian and store it columnwise in `DJAC`. The input arguments `U` and `FVAL` contain the current values of u and $F(u)$, respectively. The vectors `WK1` and `WK2`, of length `NEQ`, are provided as work space for use in `FKDJAC`. `IER` is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if `FKDJAC` failed unrecoverably (in which case the solution process is halted). NOTE: The argument `NEQ` has a type consistent with C type `long int` even in the case when the Lapack dense solver is to be used.

If the `FKDJAC` routine is provided, then, following the call to `FKINDENSE`, the user must make the call:

```
CALL FKINDENSESETJAC (FLAG, IER)
```

with `FLAG` $\neq 0$ to specify use of the user-supplied Jacobian approximation. The argument `IER` is an error return flag which is 0 for success or non-zero if an error occurred. If using the Lapack-based direct dense linear solver, the use of a Jacobian approximation supplied by the user is indicated through the call

```
CALL FKINLAPACKDENSESETJAC (FLAG, IER)
```

Optional outputs specific to the DENSE case are listed in Table 5.2.

[S] Band treatment of the linear system

To use the direct band linear solver based on the internal KINSOL implementation, the user must make the call:

```
CALL FKINBAND (NEQ, MU, ML, IER)
```

The arguments are: `MU`, the upper half-bandwidth; `ML`, the lower half-bandwidth; and `IER`, an error return flag which is 0 for success , -1 if a memory allocation failure occurred, or -2 in case an input has an illegal value.

Alternatively, to use the Lapack-based direct band linear solver, the user must make the call:

```
CALL FKINLAPACKBAND(NEQ, MU, ML, IER)
```

where the arguments have the same meanings as for `FKINBAND`, except that here `NEQ`, `MU`, and `ML` must be declared so as to match C type `int`.

As an option when using the BAND linear solver, the user may supply a routine that computes a band approximation of the system Jacobian $J = \partial F/\partial u$. If supplied, it must have the following form:

```
SUBROUTINE FKBJAC (NEQ, MU, ML, MDIM, U, FVAL, BJAC, WK1, WK2, IER)
DIMENSION U(*), FVAL(*), BJAC(MDIM,*), WK1(*), WK2(*)
```

Typically this routine will use only `NEQ`, `MU`, `ML`, `U`, and `BJAC`. It must load the `MDIM` by `N` array `BJAC` with the Jacobian matrix at the current u in band form. Store in `BJAC`(k, j) the Jacobian element $J_{i,j}$ with $k = i - j +$ `MU` $+1$ ($k = 1 \cdots$ `ML + MU + 1`) and $j = 1 \cdots N$. The input arguments `U` and `FVAL` contain the current values of u, and $F(u)$, respectively. The vectors `WK1` and `WK2` of length `NEQ` are provided as work space for use in `FKBJAC`. `IER` is an error return flag, which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if `FKBJAC` failed unrecoverably (in which case the solution process is halted). NOTE: The arguments `NEQ`, `MU`, `ML`, and `MDIM` have a type consistent with C type `long int` even in the case when the Lapack band solver is to be used.

If the `FKBJAC` routine is provided, then, following the call to `FKINBAND`, the user must make the call:

```
CALL FKINBANDSETJAC (FLAG, IER)
```

with `FLAG` $\neq 0$ to specify use of the user-supplied Jacobian approximation. The argument `IER` is an error return flag which is 0 for success or non-zero if an error occurred. If using the Lapack-based direct band linear solver, the use of a Jacobian approximation supplied by the user is indicated through the call

```
CALL FKINLAPACKNBANDSETJAC (FLAG, IER)
```

Optional outputs specific to the BAND case are listed in Table 5.2.

[S] Sparse direct treatment of the linear system

To use the KLU sparse direct linear solver, the user must make the call:

```
CALL FKINKLU (NEQ, NNZ, ORDERING, IER)
```

where `NEQ` is the size of the nonlinear system, `NNZ` is the maximum number of nonzeros in the Jacobian matrix, and `ORDERING` is the matrix ordering desired with possible values from the KLU package (0 = AMD, 1 = COLAMD). The argument `IER` is an error return flag which is 0 for success or negative for an error.

The KINSOL KLU solver will reuse much of the factorization information from one nonlinear iteration to the next. If at any time the user wants to force a full refactorization, or if the number of nonzeros in the Jacobian matrix changes, the user should make the call

```
CALL FKINKLUREINIT(NEQ, NNZ, REINIT_TYPE)
```

where NEQ is the size of the nonlinear system, NNZ is the maximum number of nonzeros in the Jacobian matrix, and REINIT_TYPE is 1 or 2. For a value of 1, the matrix will be destroyed and a new one will be allocated with NNZ nonzeros. For a value of 2, only symbolic and numeric factorizations will be completed.

Alternatively, to use the SuperLUMT linear solver, the user must make the call:

```
CALL FKINSUPERLUMT (NEQ, NNZ, ORDERING, IER)
```

where the arguments have the same meanings as for FKINKLU, except that here possible values for ORDERING derive from the SUPERLUMT package and include: 0 for Natural ordering, 1 for Minimum degree on $A^T A$, 2 for Minimum degree on $A^T + A$, and 3 for COLAMD.

If the either of the sparse direct interface packages are used, then the user must supply the FKINSPJAC routine that computes a compressed-sparse-column approximation of the system Jacobian $J = \partial F/\partial u$. If supplied, it must have the following form:

```
 SUBROUTINE FKINSPJAC(Y, FY, N, NNZ, JDATA, JRVALS,
&                     JCPTRS, WK1, WK2, IER)
```

Typically this routine will use only N, NNZ, JDATA, JRVALS and JCPTRS. It must load the N by N compressed sparse column matrix with storage for NNZ nonzeros, stored in the arrays JDATA (nonzero values), JRVALS (row indices for each nonzero), JCOLPTRS (indices for start of each column), with the Jacobian matrix at the current (y) in CSC form (see sundials_sparse.h for more information). The arguments are Y, an array containing state variables; FY, an array containing residual values; N, the number of matrix rows/columns in the Jacobian; NNZ, allocated length of nonzero storage; JDATA, nonzero values in the Jacobian (of length NNZ); JRVALS, row indices for each nonzero in Jacobian (of length NNZ); JCPTRS, pointers to each Jacobian column in the two preceding arrays (of length N+1); WK*, work arrays containing temporary workspace of same size as Y; and IER, error return code (0 if successful, >0 if a recoverable error occurred, or <0 if an unrecoverable error occurred.)

Optional outputs specific to the SPARSE case are listed in Table 5.2.

[S][P] SPGMR and SPFGMR treatment of the linear systems

For the Scaled Preconditioned GMRES or the Scaled Preconditioned Flexible GMRES solution of the linear systems, the user must make either the call

```
CALL FKINSPGMR (MAXL, MAXLRST, IER)
```

or the call

```
CALL FKINSPFGMR (MAXL, MAXLRST, IER)
```

The arguments are as follows. MAXL is the maximum Krylov subspace dimension. MAXLRST is the maximum number of restarts. IER is an error return flag which is 0 to indicate success, -1 if a memory allocation failure occurred, or -2 to indicate an illegal input.

Optional outputs specific to the SPGMR and SPFGMR cases are listed in Table 5.2.

For descriptions of the relevant optional user-supplied routines, see **User-supplied routines for SPGMR/SPFMGR/SPBCG/SPTFQMR** below.

[S][P] SPBCG treatment of the linear systems

For the Scaled Preconditioned Bi-CGStab solution of the linear systems, the user must make the call

```
CALL FKINSPBCG (MAXL, IER)
```

Its arguments are the same as those with the same names for FKINSPGMR.

Optional outputs specific to the SPBCG case are listed in Table 5.2.

For descriptions of the relevant optional user-supplied routines, see below.

[S][P] SPTFQMR treatment of the linear systems

For the Scaled Preconditioned Transpose-Free Quasi-Minimal Residual solution of the linear systems, the user must make the call

```
CALL FKINSPTFQMR (MAXL, IER)
```

Its arguments are the same as those with the same names for FKINSPGMR.

Optional outputs specific to the SPTFQMR case are listed in Table 5.2.

For descriptions of the relevant optional user-supplied routines, see below.

[S][P] Functions used by SPGMR/SPFGMR/SPBCG/SPTFQMR

An optional user-supplied routine, FKINJTIMES (see below), can be provided for Jacobian-vector products. (Note that this routine is required if Picard iteration is being used.) If it is, then, following the call to FKINSPGMR, FKINSPFGMR, FKINSPBCG, or FKINSPTFQMR, the user must make the call:

```
CALL FKINSPILSSETJAC (FLAG, IER)
```

with FLAG $\neq 0$ to specify use of the user-supplied Jacobian-times-vector approximation. The argument IER is an error return flag which is 0 for success or non-zero if an error occurred.

If preconditioning is to be done, then the user must call

```
CALL FKINSPILSSETPREC (FLAG, IER)
```

with FLAG $\neq 0$. The return flag IER is 0 if successful, or negative if a memory error occurred. In addition, the user program must include preconditioner routines FKPSOL and FKPSET (see below).

[S][P] User-supplied routines for SPGMR/SPFGMR/SPBCG/SPTFQMR

With treatment of the linear systems by any of the Krylov iterative solvers, there are three optional user-supplied routines — FKINJTIMES, FKPSOL, and FKPSET. The specifications for these routines are given below.

As an option when using the SPGMR, SPFGMR, SPBCG, or SPTFQMR linear solvers, the user may supply a routine that computes the product of the system Jacobian $J = \partial F/\partial u$ and a given vector v. If supplied, it must have the following form:

```
SUBROUTINE FKINJTIMES (V, FJV, NEWU, U, IER)
DIMENSION V(*), FJV(*), U(*)
```

Typically this routine will use only U, V, and FJV. It must compute the product vector Jv, where the vector v is stored in V, and store the product in FJV. The input argument U contains the current value of u. On return, set IER = 0 if FKINJTIMES was successful, and nonzero otherwise. NEWU is a flag to indicate if U has been changed since the last call; if it has, then NEWU = 1, and FKINJTIMES should recompute any saved Jacobian data it uses and reset NEWU to 0. (See §4.6.7.)

If preconditioning is to be included, the following routine must be supplied, for solution of the preconditioner linear system:

```
SUBROUTINE FKPSOL (U, USCALE, FVAL, FSCALE, VTEM, FTEM, IER)
DIMENSION U(*), USCALE(*), FVAL(*), FSCALE(*), VTEM(*), FTEM(*)
```

Typically this routine will use only U, FVAL, VTEM and FTEM. It must solve the preconditioned linear system $Pz = r$, where $r =$ VTEM is input, and store the solution z in VTEM as well. Here P is the right preconditioner. If scaling is being used, the routine supplied must also account for scaling on either coordinate or function value, as given in the arrays USCALE and FSCALE, respectively.

If the user's preconditioner requires that any Jacobian-related data be evaluated or preprocessed, then the following routine can be used for the evaluation and preprocessing of the preconditioner:

```
SUBROUTINE FKPSET (U, USCALE, FVAL, FSCALE, VTEMP1, VTEMP2, IER)
DIMENSION U(*), USCALE(*), FVAL(*), FSCALE(*), VTEMP1(*), VTEMP2(*)
```

It must perform any evaluation of Jacobian-related data and preprocessing needed for the solution of the preconditioned linear systems by FKPSOL. The variables U through FSCALE are for use in the preconditioning setup process. Typically, the system function FKFUN is called before any calls to FKPSET, so that FVAL will have been updated. U is the current solution iterate. The arrays VTEMP1 and VTEMP2 are available for work space. If scaling is being used, USCALE and FSCALE are available for those operations requiring scaling.

On return, set IER $= 0$ if FKPSET was successful, or set IER $= 1$ if an error occurred.

If the user calls FKINSPILSSETPREC, the routine FKPSET must be provided, even if it is not needed, and then it should return IER = 0.

6. **Problem solution**

   Solving the nonlinear system is accomplished by making the following call:

   ```
   CALL FKINSOL (U, GLOBALSTRAT, USCALE, FSCALE, IER)
   ```

   The arguments are as follows. U is an array containing the initial guess on input, and the solution on return. GLOBALSTRAT is an integer (type INTEGER) defining the global strategy choice (0 specifies Inexact Newton, 1 indicates Newton with line search, 2 indicates Picard iteration, and 3 indicates Fixed Point iteration). USCALE is an array of scaling factors for the U vector. FSCALE is an array of scaling factors for the FVAL vector. IER is an integer completion flag and will have one of the following values: 0 to indicate success, 1 to indicate that the initial guess satisfies $F(u) = 0$ within tolerances, 2 to indicate apparent stalling (small step), or a negative value to indicate an error or failure. These values correspond to the KINSol returns (see §4.5.3 and §B.2). The values of the optional outputs are available in IOPT and ROPT (see Table 5.2).

7. **Memory deallocation**

   To free the internal memory created by the call to FKINMALLOC, make the call

   ```
   CALL FKINFREE
   ```

5.5 FKINSOL optional input and output

In order to keep the number of user-callable FKINSOL interface routines to a minimum, optional inputs to the KINSOL solver are passed through only three routines: FKINSETIIN for integer optional inputs, FKINSETRIN for real optional inputs, and FKINSETVIN for real vector (array) optional inputs. These functions should be called as follows:

```
CALL FKINSETIIN (KEY, IVAL, IER)
CALL FKINSETRIN (KEY, RVAL, IER)
CALL FKINSETVIN (KEY, VVAL, IER)
```

Table 5.1: Keys for setting FKINSOL optional inputs

Integer optional inputs FKINSETIIN

Key	Optional input	Default value
PRNT_LEVEL	Verbosity level of output	0
MAX_NITER	Maximum no. of nonlinear iterations	200
ETA_FORM	Form of η coefficient	1 (KIN_ETACHOICE1)
MAX_SETUPS	Maximum no. of iterations without prec. setup	10
MAX_SP_SETUPS	Maximum no. of iterations without residual check	5
NO_INIT_SETUP	No initial preconditioner setup	FALSE
NO_MIN_EPS	Lower bound on ϵ	FALSE
NO_RES_MON	No residual monitoring	FALSE

Real optional inputs (FKINSETRIN)

Key	Optional input	Default value
FNORM_TOL	Function-norm stopping tolerance	uround$^{1/3}$
SSTEP_TOL	Scaled-step stopping tolerance	uround$^{2/3}$
MAX_STEP	Max. scaled length of Newton step	$1000\|D_u u_0\|_2$
RERR_FUNC	Relative error for F.D. Jv	$\sqrt{\text{uround}}$
ETA_CONST	Constant value of η	0.1
ETA_PARAMS	Values of γ and α	0.9 and 2.0
RMON_CONST	Constant value of ω	0.9
RMON_PARAMS	Values of ω_{min} and ω_{max}	0.00001 and 0.9

where KEY is a quoted string indicating which optional input is set, IVAL is the integer input value to be used, RVAL is the real input value to be used, and VVAL is the input real array to be used. IER is an integer return flag which is set to 0 on success and a negative value if a failure occurred. For the legal values of KEY in calls to FKINSETIIN and FKINSETRIN, see Table 5.1. The one legal value of KEY for FKINSETVIN is CONSTR_VEC, for providing the array of inequality constraints to be imposed on the solution, if any. The integer IVAL should be declared in a manner consistent with C type `long int`.

The optional outputs from the KINSOL solver are accessed not through individual functions, but rather through a pair of arrays, IOUT (integer type) of dimension at least 15, and ROUT (real type) of dimension at least 2. These arrays are owned (and allocated) by the user and are passed as arguments to FKINMALLOC. Table 5.2 lists the entries in these two arrays and specifies the optional variable as well as the KINSOL function which is actually called to extract the optional output.

For more details on the optional inputs and outputs, see §4.5.4 and §4.5.5.

5.6 Usage of the FKINBBD interface to KINBBDPRE

The FKINBBD interface sub-module is a package of C functions which, as part of the FKINSOL interface module, support the use of the KINSOL solver with the parallel NVECTOR_PARALLEL module and the KINBBDPRE preconditioner module (see §4.7), for the solution of nonlinear problems in a mixed FORTRAN/C setting.

The user-callable functions in this package, with the corresponding KINSOL and KINBBDPRE functions, are as follows:

- FKINBBDINIT interfaces to KINBBDPrecInit.
- FKINBBDOPT interfaces to KINBBDPRE optional output functions.

In addition to the FORTRAN right-hand side function FKFUN, the user-supplied functions used by this package, are listed below, each with the corresponding interface function which calls it (and its type within KINBBDPRE or KINSOL):

Table 5.2: Description of the FKINSOL optional output arrays `IOUT` and `ROUT`

Integer output array `IOUT`

Index	**Optional output**	KINSOL **function**
	KINSOL main solver	
1	`LENRW`	`KINGetWorkSpace`
2	`LENIW`	`KINGetWorkSpace`
3	`NNI`	`KINGetNumNonlinSolvIters`
4	`NFE`	`KINGetNumFuncEvals`
5	`NBCF`	`KINGetNumBetaCondFails`
6	`NBKTRK`	`KINGetNumBacktrackOps`
	KINDLS linear solvers	
7	`LENRWLS`	`KINDlsGetWorkSpace`
8	`LENIWLS`	`KINDlsGetWorkSpace`
9	`LS_FLAG`	`KINDlsGetLastFlag`
10	`NFELS`	`KINDlsGetNumFuncEvals`
11	`NJE`	`KINDlsGetNumJacEvals`
	KINSLS linear solvers	
8	`LS_FLAG`	`KINSlsGetLastFlag`
10	`NJE`	`KINSlsGetNumJacEvals`
	KINSPILS linear solvers	
7	`LENRWLS`	`KINSpilsGetWorkSpace`
8	`LENIWLS`	`KINSpilsGetWorkSpace`
9	`LS_FLAG`	`KINSpilsGetLastFlag`
10	`NFELS`	`KINSpilsGetNumFuncEvals`
11	`NJTV`	`KINSpilsGetNumJacEvals`
12	`NPE`	`KINSpilsGetNumPrecEvals`
13	`NPS`	`KINSpilsGetNumPrecSolves`
14	`NLI`	`KINSpilsGetNumLinIters`
15	`NCFL`	`KINSpilsGetNumConvFails`

Real output array `ROUT`

Index	**Optional output**	KINSOL **function**
1	`FNORM`	`KINGetFuncNorm`
2	`SSTEP`	`KINGetStepLength`

FKINBBD routine (FORTRAN, user-supplied)	KINSOL function (C, interface)	KINSOL type of interface function
FKLOCFN	FKINgloc	KINLocalFn
FKCOMMF	FKINgcomm	KINCommFn
FKJTIMES	FKINJtimes	KINSpilsJacTimesVecFn

As with the rest of the FKINSOL routines, the names of all user-supplied routines here are fixed, in order to maximize portability for the resulting mixed-language program. Additionally, based on flags discussed above in §5.3, the names of the user-supplied routines are mapped to actual values through a series of definitions in the header file `fkinbbd.h`.

The following is a summary of the usage of this module. Steps that are unchanged from the main program described in §5.4 are grayed-out.

1. **Nonlinear system function specification**

2. NVECTOR **module initialization**

3. **Problem specification**

4. **Set optional inputs**

5. **Linear solver specification**

   First, specify one of the KINSPILS iterative linear solvers, by calling one of `FKINSPGMR`, `FKINSPFGRM`, `FKINSPBCG`, or `FKINSPTFQMR`.

   To initialize the KINBBDPRE preconditioner, make the following call:

   ```
   CALL FKINBBDINIT (NLOCAL, MUDQ, MLDQ, MU, ML, IER)
   ```

   The arguments are as follows. `NLOCAL` is the local size of vectors for this process. `MUDQ` and `MLDQ` are the upper and lower half-bandwidths to be used in the computation of the local Jacobian blocks by difference quotients; these may be smaller than the true half-bandwidths of the Jacobian of the local block of G, when smaller values may provide greater efficiency. `MU` and `ML` are the upper and lower half-bandwidths of the band matrix that is retained as an approximation of the local Jacobian block; these may be smaller than `MUDQ` and `MLDQ`. `IER` is a return completion flag. A value of 0 indicates success, while a value of -1 indicates that a memory failure occurred or that an input had an illegal value.

   Optionally, to specify that SPGMR, SPFGMR, SPBCG, or SPTFQMR should use the supplied `FKJTIMES`, make the call

   ```
   CALL FKINSPILSSETJAC (FLAG, IER)
   ```

   with `FLAG` $\neq 0$. (See step 5 in §5.4).

6. **Problem solution**

7. KINBBDPRE **Optional outputs**

   Optional outputs specific to the SPGMR, SPFGMR, SPBCG, or SPTFQMR solver are listed in Table 5.2. To obtain the optional outputs associated with the KINBBDPRE module, make the following call:

   ```
   CALL FKINBBDOPT (LENRBBD, LENIBBD, NGEBBD)
   ```

   The arguments should be consistent with C type `long int`. Their returned values are as follows: `LENRBBD` is the length of real preconditioner work space, in `realtype` words. `LENIBBD` is the length of integer preconditioner work space, in integer words. These sizes are local to the current process. `NGEBBD` is the cumulative number of $G(u)$ evaluations (calls to `FKLOCFN`) so far.

8. **Memory deallocation**

   (The memory allocated for the FKINBBD module is deallocated automatically by `FKINFREE`.)

9. **User-supplied routines**

   The following two routines must be supplied for use with the KINBBDPRE module:

   ```
   SUBROUTINE FKLOCFN (NLOC, ULOC, GLOC, IER)
   DIMENSION ULOC(*), GLOC(*)
   ```

   This routine is to evaluate the function $G(u)$ approximating F (possibly identical to F), in terms of the array `ULOC` (of length `NLOC`), which is the sub-vector of u local to this processor. The resulting (local) sub-vector is to be stored in the array `GLOC`. `IER` is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if `FKLOCFN` failed unrecoverably (in which case the solution process is halted).

   ```
   SUBROUTINE FKCOMMFN (NLOC, ULOC, IER)
   DIMENSION ULOC(*)
   ```

   This routine is to perform the inter-processor communication necessary for the `FKLOCFN` routine. Each call to `FKCOMMFN` is preceded by a call to the system function routine `FKFUN` with the same argument `ULOC`. `IER` is an error return flag which should be set to 0 if successful, a positive value if a recoverable error occurred (in which case KINSOL will attempt to correct), or a negative value if `FKCOMMFN` failed recoverably (in which case the solution process is halted).

   The subroutine `FKCOMMFN` must be supplied even if it is not needed and must return `IER = 0`.

   Optionally, the user can supply a routine `FKINJTIMES` for the evaluation of Jacobian-vector products, as described above in step 5 in §5.4. Note that this routine is required if using Picard iteration.

Chapter 6

Description of the NVECTOR module

The SUNDIALS solvers are written in a data-independent manner. They all operate on generic vectors (of type `N_Vector`) through a set of operations defined by the particular NVECTOR implementation. Users can provide their own specific implementation of the NVECTOR module, or use one of four provided within SUNDIALS – a serial implementation and three parallel implementations. The generic operations are described below. In the sections following, the implementations provided with SUNDIALS are described.

The generic `N_Vector` type is a pointer to a structure that has an implementation-dependent *content* field containing the description and actual data of the vector, and an *ops* field pointing to a structure with generic vector operations. The type `N_Vector` is defined as

```
typedef struct _generic_N_Vector *N_Vector;

struct _generic_N_Vector {
    void *content;
    struct _generic_N_Vector_Ops *ops;
};
```

The `_generic_N_Vector_Ops` structure is essentially a list of pointers to the various actual vector operations, and is defined as

```
struct _generic_N_Vector_Ops {
  N_Vector    (*nvclone)(N_Vector);
  N_Vector    (*nvcloneempty)(N_Vector);
  void        (*nvdestroy)(N_Vector);
  void        (*nvspace)(N_Vector, long int *, long int *);
  realtype*   (*nvgetarraypointer)(N_Vector);
  void        (*nvsetarraypointer)(realtype *, N_Vector);
  void        (*nvlinearsum)(realtype, N_Vector, realtype, N_Vector, N_Vector);
  void        (*nvconst)(realtype, N_Vector);
  void        (*nvprod)(N_Vector, N_Vector, N_Vector);
  void        (*nvdiv)(N_Vector, N_Vector, N_Vector);
  void        (*nvscale)(realtype, N_Vector, N_Vector);
  void        (*nvabs)(N_Vector, N_Vector);
  void        (*nvinv)(N_Vector, N_Vector);
  void        (*nvaddconst)(N_Vector, realtype, N_Vector);
  realtype    (*nvdotprod)(N_Vector, N_Vector);
  realtype    (*nvmaxnorm)(N_Vector);
  realtype    (*nvwrmsnorm)(N_Vector, N_Vector);
  realtype    (*nvwrmsnormmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvmin)(N_Vector);
  realtype    (*nvwl2norm)(N_Vector, N_Vector);
  realtype    (*nvl1norm)(N_Vector);
  void        (*nvcompare)(realtype, N_Vector, N_Vector);
  booleantype (*nvinvtest)(N_Vector, N_Vector);
  booleantype (*nvconstrmask)(N_Vector, N_Vector, N_Vector);
  realtype    (*nvminquotient)(N_Vector, N_Vector);
};
```

The generic NVECTOR module defines and implements the vector operations acting on `N_Vector`. These routines are nothing but wrappers for the vector operations defined by a particular NVECTOR implementation, which are accessed through the *ops* field of the `N_Vector` structure. To illustrate this point we show below the implementation of a typical vector operation from the generic NVECTOR module, namely `N_VScale`, which performs the scaling of a vector `x` by a scalar `c`:

```
void N_VScale(realtype c, N_Vector x, N_Vector z)
{
   z->ops->nvscale(c, x, z);
}
```

Table 6.1 contains a complete list of all vector operations defined by the generic NVECTOR module.

Finally, note that the generic NVECTOR module defines the functions `N_VCloneVectorArray` and `N_VCloneEmptyVectorArray`. Both functions create (by cloning) an array of `count` variables of type `N_Vector`, each of the same type as an existing `N_Vector`. Their prototypes are

```
N_Vector *N_VCloneVectorArray(int count, N_Vector w);
N_Vector *N_VCloneEmptyVectorArray(int count, N_Vector w);
```

and their definitions are based on the implementation-specific `N_VClone` and `N_VCloneEmpty` operations, respectively.

An array of variables of type `N_Vector` can be destroyed by calling `N_VDestroyVectorArray`, whose prototype is

```
void N_VDestroyVectorArray(N_Vector *vs, int count);
```

and whose definition is based on the implementation-specific `N_VDestroy` operation.

A particular implementation of the NVECTOR module must:

- Specify the *content* field of `N_Vector`.
- Define and implement the vector operations. Note that the names of these routines should be unique to that implementation in order to permit using more than one NVECTOR module (each with different `N_Vector` internal data representations) in the same code.
- Define and implement user-callable constructor and destructor routines to create and free an `N_Vector` with the new *content* field and with *ops* pointing to the new vector operations.
- Optionally, define and implement additional user-callable routines acting on the newly defined `N_Vector` (e.g., a routine to print the content for debugging purposes).
- Optionally, provide accessor macros as needed for that particular implementation to be used to access different parts in the *content* field of the newly defined `N_Vector`.

Table 6.1: Description of the NVECTOR operations

Name	**Usage and Description**
`N_VClone`	`v = N_VClone(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not copy the vector, but rather allocates storage for the new vector.
`N_VCloneEmpty`	`v = N_VCloneEmpty(w);` Creates a new `N_Vector` of the same type as an existing vector `w` and sets the *ops* field. It does not allocate storage for data.
`N_VDestroy`	`N_VDestroy(v);` Destroys the `N_Vector v` and frees memory allocated for its internal data.
`N_VSpace`	`N_VSpace(nvSpec, &lrw, &liw);` Returns storage requirements for one `N_Vector`. `lrw` contains the number of realtype words and `liw` contains the number of integer words. This function is advisory only, for use in determining a user's total space requirements; it could be a dummy function in a user-supplied NVECTOR module if that information is not of interest.
`N_VGetArrayPointer`	`vdata = N_VGetArrayPointer(v);` Returns a pointer to a `realtype` array from the `N_Vector v`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the solver-specific interfaces to the dense and banded (serial) linear solvers, the sparse linear solvers (serial and threaded), and in the interfaces to the banded (serial) and band-block-diagonal (parallel) preconditioner modules provided with SUNDIALS.
`N_VSetArrayPointer`	`N_VSetArrayPointer(vdata, v);` Overwrites the data in an `N_Vector` with a given array of `realtype`. Note that this assumes that the internal data in `N_Vector` is a contiguous array of `realtype`. This routine is only used in the interfaces to the dense (serial) linear solver, hence need not exist in a user-supplied NVECTOR module for a parallel environment.
`N_VLinearSum`	`N_VLinearSum(a, x, b, y, z);` Performs the operation $z = ax + by$, where a and b are `realtype` scalars and x and y are of type `N_Vector`: $z_i = ax_i + by_i$, $i = 0, \ldots, n-1$.
`N_VConst`	`N_VConst(c, z);` Sets all components of the `N_Vector z` to `realtype c`: $z_i = c$, $i = 0, \ldots, n-1$.

continued on next page

continued from last page

Name	**Usage and Description**
`N_VProd`	`N_VProd(x, y, z);` Sets the `N_Vector` `z` to be the component-wise product of the `N_Vector` inputs `x` and `y`: $z_i = x_i y_i,\ i = 0, \dots, n-1$.
`N_VDiv`	`N_VDiv(x, y, z);` Sets the `N_Vector` `z` to be the component-wise ratio of the `N_Vector` inputs `x` and `y`: $z_i = x_i / y_i,\ i = 0, \dots, n-1$. The y_i may not be tested for 0 values. It should only be called with a `y` that is guaranteed to have all nonzero components.
`N_VScale`	`N_VScale(c, x, z);` Scales the `N_Vector` `x` by the `realtype` scalar `c` and returns the result in `z`: $z_i = c x_i,\ i = 0, \dots, n-1$.
`N_VAbs`	`N_VAbs(x, z);` Sets the components of the `N_Vector` `z` to be the absolute values of the components of the `N_Vector` `x`: $y_i = \lvert x_i \rvert,\ i = 0, \dots, n-1$.
`N_VInv`	`N_VInv(x, z);` Sets the components of the `N_Vector` `z` to be the inverses of the components of the `N_Vector` `x`: $z_i = 1.0/x_i,\ i = 0, \dots, n-1$. This routine may not check for division by 0. It should be called only with an `x` which is guaranteed to have all nonzero components.
`N_VAddConst`	`N_VAddConst(x, b, z);` Adds the `realtype` scalar `b` to all components of `x` and returns the result in the `N_Vector` `z`: $z_i = x_i + b,\ i = 0, \dots, n-1$.
`N_VDotProd`	`d = N_VDotProd(x, y);` Returns the value of the ordinary dot product of `x` and `y`: $d = \sum_{i=0}^{n-1} x_i y_i$.
`N_VMaxNorm`	`m = N_VMaxNorm(x);` Returns the maximum norm of the `N_Vector` `x`: $m = \max_i \lvert x_i \rvert$.
`N_VWrmsNorm`	`m = N_VWrmsNorm(x, w)` Returns the weighted root-mean-square norm of the `N_Vector` `x` with `realtype` weight vector `w`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i)^2 \right) / n}$.
`N_VWrmsNormMask`	`m = N_VWrmsNormMask(x, w, id);` Returns the weighted root mean square norm of the `N_Vector` `x` with `realtype` weight vector `w` built using only the elements of `x` corresponding to nonzero elements of the `N_Vector` `id`: $m = \sqrt{\left(\sum_{i=0}^{n-1} (x_i w_i \operatorname{sign}(id_i))^2 \right) / n}$.
`N_VMin`	`m = N_VMin(x);` Returns the smallest element of the `N_Vector` `x`: $m = \min_i x_i$.

continued on next page

continued from last page

Name	**Usage and Description**
`N_VWL2Norm`	`m = N_VWL2Norm(x, w);` Returns the weighted Euclidean ℓ_2 norm of the `N_Vector x` with `realtype` weight vector `w`: $m = \sqrt{\sum_{i=0}^{n-1}(x_i w_i)^2}$.
`N_VL1Norm`	`m = N_VL1Norm(x);` Returns the ℓ_1 norm of the `N_Vector x`: $m = \sum_{i=0}^{n-1}\|x_i\|$.
`N_VCompare`	`N_VCompare(c, x, z);` Compares the components of the `N_Vector x` to the `realtype` scalar `c` and returns an `N_Vector z` such that: $z_i = 1.0$ if $\|x_i\| \geq c$ and $z_i = 0.0$ otherwise.
`N_VInvTest`	`t = N_VInvTest(x, z);` Sets the components of the `N_Vector z` to be the inverses of the components of the `N_Vector x`, with prior testing for zero values: $z_i = 1.0/x_i$, $i = 0, \ldots, n-1$. This routine returns a boolean assigned to `TRUE` if all components of `x` are nonzero (successful inversion) and returns `FALSE` otherwise.
`N_VConstrMask`	`t = N_VConstrMask(c, x, m);` Performs the following constraint tests: $x_i > 0$ if $c_i = 2$, $x_i \geq 0$ if $c_i = 1$, $x_i \leq 0$ if $c_i = -1$, $x_i < 0$ if $c_i = -2$. There is no constraint on x_i if $c_i = 0$. This routine returns a boolean assigned to `FALSE` if any element failed the constraint test and assigned to `TRUE` if all passed. It also sets a mask vector `m`, with elements equal to 1.0 where the constraint test failed, and 0.0 where the test passed. This routine is used only for constraint checking.
`N_VMinQuotient`	`minq = N_VMinQuotient(num, denom);` This routine returns the minimum of the quotients obtained by termwise dividing $\texttt{num}_i$ by $\texttt{denom}_i$. A zero element in `denom` will be skipped. If no such quotients are found, then the large value `BIG_REAL` (defined in the header file `sundials_types.h`) is returned.

6.1 The NVECTOR_SERIAL implementation

The serial implementation of the NVECTOR module provided with SUNDIALS, NVECTOR_SERIAL, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, and a boolean flag *own_data* which specifies the ownership of *data*.

```
struct _N_VectorContent_Serial {
  long int length;
  booleantype own_data;
  realtype *data;
};
```

The following five macros are provided to access the content of an NVECTOR_SERIAL vector. The suffix `_S` in the names denotes the serial version.

- `NV_CONTENT_S`

  This routine gives access to the contents of the serial vector `N_Vector`.

The assignment `v_cont` = `NV_CONTENT_S(v)` sets `v_cont` to be a pointer to the serial `N_Vector` content structure.

Implementation:

```
#define NV_CONTENT_S(v) ( (N_VectorContent_Serial)(v->content) )
```

- `NV_OWN_DATA_S`, `NV_DATA_S`, `NV_LENGTH_S`

  These macros give individual access to the parts of the content of a serial `N_Vector`.

  The assignment `v_data = NV_DATA_S(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_S(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_len = NV_LENGTH_S(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_S(v) = len_v` sets the length of `v` to be `len_v`.

  Implementation:

  ```
  #define NV_OWN_DATA_S(v) ( NV_CONTENT_S(v)->own_data )
  #define NV_DATA_S(v) ( NV_CONTENT_S(v)->data )
  #define NV_LENGTH_S(v) ( NV_CONTENT_S(v)->length )
  ```

- `NV_Ith_S`

  This macro gives access to the individual components of the data array of an `N_Vector`.

  The assignment `r = NV_Ith_S(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_S(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_S(v,i) ( NV_DATA_S(v)[i] )
  ```

The NVECTOR_SERIAL module defines serial implementations of all vector operations listed in Table 6.1. Their names are obtained from those in Table 6.1 by appending the suffix `_Serial`. The module NVECTOR_SERIAL provides the following additional user-callable routines:

- `N_VNew_Serial`

  This function creates and allocates memory for a serial `N_Vector`. Its only argument is the vector length.

  ```
  N_Vector N_VNew_Serial(long int vec_length);
  ```

- `N_VNewEmpty_Serial`

  This function creates a new serial `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_Serial(long int vec_length);
  ```

- `N_VMake_Serial`

  This function creates and allocates memory for a serial vector with user-provided data array.

  ```
  N_Vector N_VMake_Serial(long int vec_length, realtype *v_data);
  ```

- `N_VCloneVectorArray_Serial`

  This function creates (by cloning) an array of `count` serial vectors.

  ```
  N_Vector *N_VCloneVectorArray_Serial(int count, N_Vector w);
  ```

- `N_VCloneEmptyVectorArray_Serial`

  This function creates (by cloning) an array of `count` serial vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneEmptyVectorArray_Serial(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Serial`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Serial` or with `N_VCloneEmptyVectorArray_Serial`.

  ```
  void N_VDestroyVectorArray_Serial(N_Vector *vs, int count);
  ```

- `N_VPrint_Serial`

  This function prints the content of a serial vector to `stdout`.

  ```
  void N_VPrint_Serial(N_Vector v);
  ```

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_S(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_S(v,i)` within the loop.
- `N_VNewEmpty_Serial`, `N_VMake_Serial`, and `N_VCloneEmptyVectorArray_Serial` set the field *own_data* = `FALSE`. `N_VDestroy_Serial` and `N_VDestroyVectorArray_Serial` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_SERIAL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.2 The NVECTOR_PARALLEL implementation

The NVECTOR_PARALLEL implementation of the NVECTOR module provided with SUNDIALS is based on MPI. It defines the *content* field of `N_Vector` to be a structure containing the global and local lengths of the vector, a pointer to the beginning of a contiguous local data array, an MPI communicator, an a boolean flag *own_data* indicating ownership of the data array *data*.

```
struct _N_VectorContent_Parallel {
  long int local_length;
  long int global_length;
  booleantype own_data;
  realtype *data;
  MPI_Comm comm;
};
```

The following seven macros are provided to access the content of a NVECTOR_PARALLEL vector. The suffix _P in the names denotes the distributed memory parallel version.

- `NV_CONTENT_P`

  This macro gives access to the contents of the parallel vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_P(v)` sets `v_cont` to be a pointer to the `N_Vector` content structure of type `struct _N_VectorParallelContent`.

  Implementation:

  ```
  #define NV_CONTENT_P(v) ( (N_VectorContent_Parallel)(v->content) )
  ```

- `NV_OWN_DATA_P`, `NV_DATA_P`, `NV_LOCLENGTH_P`, `NV_GLOBLENGTH_P`

  These macros give individual access to the parts of the content of a parallel `N_Vector`.

The assignment `v_data = NV_DATA_P(v)` sets `v_data` to be a pointer to the first component of the local data for the `N_Vector v`. The assignment `NV_DATA_P(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

The assignment `v_llen = NV_LOCLENGTH_P(v)` sets `v_llen` to be the length of the local part of `v`. The call `NV_LENGTH_P(v) = llen_v` sets the local length of `v` to be `llen_v`.

The assignment `v_glen = NV_GLOBLENGTH_P(v)` sets `v_glen` to be the global length of the vector `v`. The call `NV_GLOBLENGTH_P(v) = glen_v` sets the global length of `v` to be `glen_v`.

Implementation:

```
#define NV_OWN_DATA_P(v)   ( NV_CONTENT_P(v)->own_data )
#define NV_DATA_P(v)       ( NV_CONTENT_P(v)->data )
#define NV_LOCLENGTH_P(v)  ( NV_CONTENT_P(v)->local_length )
#define NV_GLOBLENGTH_P(v) ( NV_CONTENT_P(v)->global_length )
```

- `NV_COMM_P`

  This macro provides access to the MPI communicator used by the NVECTOR_PARALLEL vectors.

  Implementation:

  ```
  #define NV_COMM_P(v) ( NV_CONTENT_P(v)->comm )
  ```

- `NV_Ith_P`

  This macro gives access to the individual components of the local data array of an `N_Vector`.

  The assignment `r = NV_Ith_P(v,i)` sets `r` to be the value of the `i`-th component of the local part of `v`. The assignment `NV_Ith_P(v,i) = r` sets the value of the `i`-th component of the local part of `v` to be `r`.

  Here i ranges from 0 to $n-1$, where n is the local length.

  Implementation:

  ```
  #define NV_Ith_P(v,i) ( NV_DATA_P(v)[i] )
  ```

The NVECTOR_PARALLEL module defines parallel implementations of all vector operations listed in Table 6.1 Their names are obtained from those in Table 6.1 by appending the suffix `_Parallel`. The module NVECTOR_PARALLEL provides the following additional user-callable routines:

- `N_VNew_Parallel`

  This function creates and allocates memory for a parallel vector.

  ```
  N_Vector N_VNew_Parallel(MPI_Comm comm,
                           long int local_length,
                           long int global_length);
  ```

- `N_VNewEmpty_Parallel`

  This function creates a new parallel `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_Parallel(MPI_Comm comm,
                                long int local_length,
                                long int global_length);
  ```

- `N_VMake_Parallel`

  This function creates and allocates memory for a parallel vector with user-provided data array.

```
N_Vector N_VMake_Parallel(MPI_Comm comm,
                          long int local_length,
                          long int global_length,
                          realtype *v_data);
```

- `N_VCloneVectorArray_Parallel`

  This function creates (by cloning) an array of `count` parallel vectors.

  ```
  N_Vector *N_VCloneVectorArray_Parallel(int count, N_Vector w);
  ```

- `N_VCloneEmptyVectorArray_Parallel`

  This function creates (by cloning) an array of `count` parallel vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneEmptyVectorArray_Parallel(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Parallel`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Parallel` or with `N_VCloneEmptyVectorArray_Parallel`.

  ```
  void N_VDestroyVectorArray_Parallel(N_Vector *vs, int count);
  ```

- `N_VPrint_Parallel`

  This function prints the content of a parallel vector to stdout.

  ```
  void N_VPrint_Parallel(N_Vector v);
  ```

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the local component array via `v_data = NV_DATA_P(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_P(v,i)` within the loop.
- `N_VNewEmpty_Parallel`, `N_VMake_Parallel`, and `N_VCloneEmptyVectorArray_Parallel` set the field *own_data* = `FALSE`. `N_VDestroy_Parallel` and `N_VDestroyVectorArray_Parallel` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.
- To maximize efficiency, vector operations in the NVECTOR_PARALLEL implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.3 The NVECTOR_OPENMP implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least $100,000$ before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The OpenMP NVECTOR implementation provided with SUNDIALS, NVECTOR_OPENMP, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using OpenMP.

```
struct _N_VectorContent_OpenMP {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The following six macros are provided to access the content of an NVECTOR_OPENMP vector. The suffix `_OMP` in the names denotes the OpenMP version.

- `NV_CONTENT_OMP`

  This routine gives access to the contents of the OpenMP vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_OMP(v)` sets `v_cont` to be a pointer to the OpenMP `N_Vector` content structure.

  Implementation:

  ```
  #define NV_CONTENT_OMP(v) ( (N_VectorContent_OpenMP)(v->content) )
  ```

- `NV_OWN_DATA_OMP`, `NV_DATA_OMP`, `NV_LENGTH_OMP`, `NV_NUM_THREADS_OMP`

  These macros give individual access to the parts of the content of a OpenMP `N_Vector`.

  The assignment `v_data = NV_DATA_OMP(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector` `v`. The assignment `NV_DATA_OMP(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_len = NV_LENGTH_OMP(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_OMP(v) = len_v` sets the length of `v` to be `len_v`.

  The assignment `v_num_threads = NV_NUM_THREADS_OMP(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_OMP(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

  Implementation:

  ```
  #define NV_OWN_DATA_OMP(v) ( NV_CONTENT_OMP(v)->own_data )

  #define NV_DATA_OMP(v) ( NV_CONTENT_OMP(v)->data )

  #define NV_LENGTH_OMP(v) ( NV_CONTENT_OMP(v)->length )

  #define NV_NUM_THREADS_OMP(v) ( NV_CONTENT_OMP(v)->num_threads )
  ```

- `NV_Ith_OMP`

  This macro gives access to the individual components of the data array of an `N_Vector`.

  The assignment `r = NV_Ith_OMP(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_OMP(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_OMP(v,i) ( NV_DATA_OMP(v)[i] )
  ```

The NVECTOR_OPENMP module defines OpenMP implementations of all vector operations listed in Table 6.1. Their names are obtained from those in Table 6.1 by appending the suffix `_OpenMP`. The module NVECTOR_OPENMP provides the following additional user-callable routines:

- `N_VNew_OpenMP`

  This function creates and allocates memory for a OpenMP `N_Vector`. Arguments are the vector length and number of threads.

  ```
  N_Vector N_VNew_OpenMP(long int vec_length, int num_threads);
  ```

- `N_VNewEmpty_OpenMP`

  This function creates a new OpenMP `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_OpenMP(long int vec_length, int num_threads);
  ```

- `N_VMake_OpenMP`

  This function creates and allocates memory for a OpenMP vector with user-provided data array.

  ```
  N_Vector N_VMake_OpenMP(long int vec_length, realtype *v_data, int num_threads);
  ```

- `N_VCloneVectorArray_OpenMP`

  This function creates (by cloning) an array of `count` OpenMP vectors.

  ```
  N_Vector *N_VCloneVectorArray_OpenMP(int count, N_Vector w);
  ```

- `N_VCloneEmptyVectorArray_OpenMP`

  This function creates (by cloning) an array of `count` OpenMP vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneEmptyVectorArray_OpenMP(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_OpenMP`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_OpenMP` or with `N_VCloneEmptyVectorArray_OpenMP`.

  ```
  void N_VDestroyVectorArray_OpenMP(N_Vector *vs, int count);
  ```

- `N_VPrint_OpenMP`

  This function prints the content of a OpenMP vector to `stdout`.

  ```
  void N_VPrint_OpenMP(N_Vector v);
  ```

Notes

- When looping over the components of an `N_Vector v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_OMP(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_OMP(v,i)` within the loop.

- `N_VNewEmpty_OpenMP`, `N_VMake_OpenMP`, and `N_VCloneEmptyVectorArray_OpenMP` set the field *own_data* = `FALSE`. `N_VDestroy_OpenMP` and `N_VDestroyVectorArray_OpenMP` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_OPENMP implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.4 The NVECTOR_PTHREADS implementation

In situations where a user has a multi-core processing unit capable of running multiple parallel threads with shared memory, SUNDIALS provides an implementation of NVECTOR using OpenMP, called NVECTOR_OPENMP, and an implementation using Pthreads, called NVECTOR_PTHREADS. Testing has shown that vectors should be of length at least $100,000$ before the overhead associated with creating and using the threads is made up by the parallelism in the vector calculations.

The Pthreads NVECTOR implementation provided with SUNDIALS, NVECTOR_PTHREADS, defines the *content* field of `N_Vector` to be a structure containing the length of the vector, a pointer to the beginning of a contiguous data array, a boolean flag *own_data* which specifies the ownership of *data*, and the number of threads. Operations on the vector are threaded using POSIX threads (Pthreads).

```
struct _N_VectorContent_Pthreads {
  long int length;
  booleantype own_data;
  realtype *data;
  int num_threads;
};
```

The following six macros are provided to access the content of an NVECTOR_PTHREADS vector. The suffix `_PT` in the names denotes the Pthreads version.

- `NV_CONTENT_PT`

  This routine gives access to the contents of the Pthreads vector `N_Vector`.

  The assignment `v_cont = NV_CONTENT_PT(v)` sets `v_cont` to be a pointer to the Pthreads `N_Vector` content structure.

  Implementation:

  ```
  #define NV_CONTENT_PT(v) ( (N_VectorContent_Pthreads)(v->content) )
  ```

- `NV_OWN_DATA_PT`, `NV_DATA_PT`, `NV_LENGTH_PT`, `NV_NUM_THREADS_PT`

  These macros give individual access to the parts of the content of a Pthreads `N_Vector`.

  The assignment `v_data = NV_DATA_PT(v)` sets `v_data` to be a pointer to the first component of the data for the `N_Vector v`. The assignment `NV_DATA_PT(v) = v_data` sets the component array of `v` to be `v_data` by storing the pointer `v_data`.

  The assignment `v_len = NV_LENGTH_PT(v)` sets `v_len` to be the length of `v`. On the other hand, the call `NV_LENGTH_PT(v) = len_v` sets the length of `v` to be `len_v`.

  The assignment `v_num_threads = NV_NUM_THREADS_PT(v)` sets `v_num_threads` to be the number of threads from `v`. On the other hand, the call `NV_NUM_THREADS_PT(v) = num_threads_v` sets the number of threads for `v` to be `num_threads_v`.

  Implementation:

  ```
  #define NV_OWN_DATA_PT(v) ( NV_CONTENT_PT(v)->own_data )

  #define NV_DATA_PT(v) ( NV_CONTENT_PT(v)->data )

  #define NV_LENGTH_PT(v) ( NV_CONTENT_PT(v)->length )

  #define NV_NUM_THREADS_PT(v) ( NV_CONTENT_PT(v)->num_threads )
  ```

- `NV_Ith_PT`

  This macro gives access to the individual components of the data array of an `N_Vector`.

  The assignment `r = NV_Ith_PT(v,i)` sets `r` to be the value of the `i`-th component of `v`. The assignment `NV_Ith_PT(v,i) = r` sets the value of the `i`-th component of `v` to be `r`.

  Here i ranges from 0 to $n-1$ for a vector of length n.

  Implementation:

  ```
  #define NV_Ith_PT(v,i) ( NV_DATA_PT(v)[i] )
  ```

The NVECTOR_PTHREADS module defines Pthreads implementations of all vector operations listed in Table 6.1. Their names are obtained from those in Table 6.1 by appending the suffix `_Pthreads`. The module NVECTOR_PTHREADS provides the following additional user-callable routines:

- `N_VNew_Pthreads`

  This function creates and allocates memory for a Pthreads `N_Vector`. Arguments are the vector length and number of threads.

  ```
  N_Vector N_VNew_Pthreads(long int vec_length, int num_threads);
  ```

- `N_VNewEmpty_Pthreads`

  This function creates a new Pthreads `N_Vector` with an empty (`NULL`) data array.

  ```
  N_Vector N_VNewEmpty_Pthreads(long int vec_length, int num_threads);
  ```

- `N_VMake_Pthreads`

  This function creates and allocates memory for a Pthreads vector with user-provided data array.

  ```
  N_Vector N_VMake_Pthreads(long int vec_length, realtype *v_data, int num_threads);
  ```

- `N_VCloneVectorArray_Pthreads`

  This function creates (by cloning) an array of `count` Pthreads vectors.

  ```
  N_Vector *N_VCloneVectorArray_Pthreads(int count, N_Vector w);
  ```

- `N_VCloneEmptyVectorArray_Pthreads`

  This function creates (by cloning) an array of `count` Pthreads vectors, each with an empty (`NULL`) data array.

  ```
  N_Vector *N_VCloneEmptyVectorArray_Pthreads(int count, N_Vector w);
  ```

- `N_VDestroyVectorArray_Pthreads`

  This function frees memory allocated for the array of `count` variables of type `N_Vector` created with `N_VCloneVectorArray_Pthreads` or with `N_VCloneEmptyVectorArray_Pthreads`.

  ```
  void N_VDestroyVectorArray_Pthreads(N_Vector *vs, int count);
  ```

- `N_VPrint_Pthreads`

  This function prints the content of a Pthreads vector to `stdout`.

  ```
  void N_VPrint_Pthreads(N_Vector v);
  ```

Notes

- When looping over the components of an `N_Vector` `v`, it is more efficient to first obtain the component array via `v_data = NV_DATA_PT(v)` and then access `v_data[i]` within the loop than it is to use `NV_Ith_PT(v,i)` within the loop.

- `N_VNewEmpty_Pthreads`, `N_VMake_Pthreads`, and `N_VCloneEmptyVectorArray_Pthreads` set the field *own_data* = `FALSE`. `N_VDestroy_Pthreads` and `N_VDestroyVectorArray_Pthreads` will not attempt to free the pointer *data* for any `N_Vector` with *own_data* set to `FALSE`. In such a case, it is the user's responsibility to deallocate the *data* pointer.

- To maximize efficiency, vector operations in the NVECTOR_PTHREADS implementation that have more than one `N_Vector` argument do not check for consistent internal representation of these vectors. It is the user's responsibility to ensure that such routines are called with `N_Vector` arguments that were all created with the same internal representations.

6.5 NVECTOR Examples

There are `NVector` examples that may be installed for each implementation: serial, parallel, OpenMP, and Pthreads. Each implementation makes use of the functions in `test_nvector.c`. These example functions show simple usage of the `NVector` family of functions. The input to the examples are the vector length, number of threads (if threaded implementation), and a print timing flag.
The following is a list of the example functions in `test_nvector.c`:

- `Test_N_VClone`: Creates clone of vector and checks validity of clone.
- `Test_N_VCloneEmpty`: Creates clone of empty vector and checks validity of clone.

- `Test_N_VCloneVectorArray`: Creates clone of vector array and checks validity of cloned array.
- `Test_N_VCloneVectorArray`: Creates clone of empty vector array and checks validity of cloned array.
- `Test_N_VGetArrayPointer`: Get array pointer.
- `Test_N_VSetArrayPointer`: Allocate new vector, set pointer to new vector array, and check values.
- `Test_N_VLinearSum` Case 1a: Test y = x + y
- `Test_N_VLinearSum` Case 1b: Test y = -x + y
- `Test_N_VLinearSum` Case 1c: Test y = ax + y
- `Test_N_VLinearSum` Case 2a: Test x = x + y
- `Test_N_VLinearSum` Case 2b: Test x = x - y
- `Test_N_VLinearSum` Case 2c: Test x = x + by
- `Test_N_VLinearSum` Case 3: Test z = x + y
- `Test_N_VLinearSum` Case 4a: Test z = x - y
- `Test_N_VLinearSum` Case 4b: Test z = -x + y
- `Test_N_VLinearSum` Case 5a: Test z = x + by
- `Test_N_VLinearSum` Case 5b: Test z = ax + y
- `Test_N_VLinearSum` Case 6a: Test z = -x + by
- `Test_N_VLinearSum` Case 6b: Test z = ax - y
- `Test_N_VLinearSum` Case 7: Test z = a(x + y)
- `Test_N_VLinearSum` Case 8: Test z = a(x - y)
- `Test_N_VLinearSum` Case 9: Test z = ax + by
- `Test_N_VConst`: Fill vector with constant and check result.
- `Test_N_VProd`: Test vector multiply: z = x * y
- `Test_N_VDiv`: Test vector division: z = x / y
- `Test_N_VScale`: Case 1: scale: x = cx
- `Test_N_VScale`: Case 2: copy: z = x
- `Test_N_VScale`: Case 3: negate: z = -x
- `Test_N_VScale`: Case 4: combination: z = cx
- `Test_N_VAbs`: Create absolute value of vector.
- `Test_N_VAddConst`: add constant vector: z = c + x
- `Test_N_VDotProd`: Calculate dot product of two vectors.
- `Test_N_VMaxNorm`: Create vector with known values, find and validate max norm.
- `Test_N_VWrmsNorm`: Create vector of known values, find and validate weighted root mean square.

- `Test_N_VWrmsNormMask`: Case 1: Create vector of known values, find and validate weighted root mean square using all elements.
- `Test_N_VWrmsNormMask`: Case 2: Create vector of known values, find and validate weighted root mean square using no elements.
- `Test_N_VMin`: Create vector, find and validate the min.
- `Test_N_VWL2Norm`: Create vector, find and validate the weighted Euclidean L2 norm.
- `Test_N_VL1Norm`: Create vector, find and validate the L1 norm.
- `Test_N_VCompare`: Compare vector with constant returning and validating comparison vector.
- `Test_N_VInvTest`: Test z[i] = 1 / x[i]
- `Test_N_VConstrMask`: Test mask of vector x with vector c.
- `Test_N_VMinQuotient`: Fill two vectors with known values. Calculate and validate minimum quotient.

6.6 NVECTOR functions used by KINSOL

In Table 6.2 below, we list the vector functions in the NVECTOR module used within the KINSOL package. The table also shows, for each function, which of the code modules uses the function. The KINSOL column shows function usage within the main solver module, while the remaining five columns show function usage within each of the seven KINSOL linear solvers, the KINBBDPRE preconditioner module, and the FKINSOL module. Here KINDLS stands for KINDENSE and KINBAND; KINSPILS stands for KINSPGMR, KINSPFGMR, KINSPBCG, and KINSPTFQMR; and KINSLS stands for KINKLU and KINSUPERLUMT.

There is one subtlety in the KINSPILS column hidden by the table, explained here for the case of the KINSPGMR module. The `N_VDotProd` function is called both within the interface file `kinsol_spgmr.c` and within the implementation files `sundials_spgmr.c` and `sundials_iterative.c` for the generic SPGMR solver upon which the KINSPGMR solver is built. Also, although `N_VDiv` and `N_VProd` are not called within the interface file `kinsol_spgmr.c`, they are called within the implementation file `sundials_spgmr.c`, and so are required by the KINSPGMR solver module. Analogous statements apply to the KINSPFGMR, KINSPBCG and KINSPTFQMR modules, except that the latter two do not use `sundials_iterative.c`. This issue does not arise for the direct KINSOL linear solvers because the generic DENSE and BAND solvers (used in the implementation of KINDENSE and KINBAND) do not make calls to any vector functions.

At this point, we should emphasize that the KINSOL user does not need to know anything about the usage of vector functions by the KINSOL code modules in order to use KINSOL. The information is presented as an implementation detail for the interested reader.

The vector functions listed in Table 6.1 that are *not* used by KINSOL are: `N_VAddConst`, `N_VWrmsNorm`, `N_VWrmsNormMask`, `N_VCompare`, and `N_VInvTest`. Therefore a user-supplied NVECTOR module for KINSOL could omit these five functions.

Table 6.2: List of vector functions usage by KINSOL code modules

	KINSOL	KINDLS	KINSPILS	KINSLS	KINBBDPRE	FKINSOL
N_VClone	✓		✓		✓	
N_VCloneEmpty						✓
N_VDestroy	✓		✓		✓	✓
N_VSpace	✓					
N_VGetArrayPointer		✓		✓	✓	✓
N_VSetArrayPointer		✓				✓
N_VLinearSum	✓	✓	✓			
N_VConst			✓			
N_VProd	✓	✓	✓	✓		
N_VDiv	✓		✓			
N_VScale	✓	✓	✓	✓	✓	
N_VAbs	✓					
N_VInv	✓					
N_VDotProd	✓	✓	✓	✓		
N_VMaxNorm	✓					
N_VMin	✓					
N_VWL2Norm	✓		✓			
N_VL1Norm			✓			
N_VConstrMask	✓					
N_VMinQuotient	✓					

Chapter 7

Providing Alternate Linear Solver Modules

The central KINSOL module interfaces with a linear solver module by way of calls to four functions. These are denoted here by `linit`, `lsetup`, `lsolve`, and `lfree`. Briefly, their purposes are as follows:

- `linit`: initialize memory specific to the linear solver;
- `lsetup`: evaluate and preprocess the Jacobian or preconditioner;
- `lsolve`: solve the linear system;
- `lfree`: free the linear solver memory.

A linear solver module must also provide a user-callable **specification function** (like that described in §4.5.2) which will attach the above four functions to the main KINSOL memory block. The KINSOL memory block is a structure defined in the header file `kinsol_impl.h`. A pointer to such a structure is defined as the type `KINMem`. The four fields in a `KINMem` structure that must point to the linear solver's functions are `kin_linit`, `kin_lsetup`, `kin_lsolve`, and `kin_lfree`, respectively. Note that of the four interface functions, only the `lsolve` function is required. The `lfree` function must be provided only if the solver specification function makes any memory allocation. For any of the functions that are *not* provided, the corresponding field should be set to `NULL`. The linear solver specification function must also set the value of the field `kin_setupNonNull` in the KINSOL memory block — to `TRUE` if `lsetup` is used, or `FALSE` otherwise.

Typically, the linear solver will require a block of memory specific to the solver, and a principal function of the specification function is to allocate that memory block, and initialize it. Then the field `kin_lmem` in the KINSOL memory block is available to attach a pointer to that linear solver memory. This block can then be used to facilitate the exchange of data between the four interface functions.

If the linear solver involves adjustable parameters, the specification function should set the default values of those. User-callable functions may be defined that could, optionally, override the default parameter values.

We encourage the use of performance counters in connection with the various operations involved with the linear solver. Such counters would be members of the linear solver memory block, would be initialized in the `linit` function, and would be incremented by the `lsetup` and `lsolve` functions. Then, user-callable functions would be needed to obtain the values of these counters.

For consistency with the existing KINSOL linear solver modules, we recommend that the return value of the specification function be 0 for a successful return, and a negative value if an error occurs. Possible error conditions include: the pointer to the main KINSOL memory block is `NULL`, an input is illegal, the NVECTOR implementation is not compatible, or a memory allocation fails.

These four functions, which interface between KINSOL and the linear solver module, necessarily have fixed call sequences. Thus a user wishing to implement another linear solver within the KINSOL package must adhere to this set of interfaces. The following is a complete description of the call list for each of these functions. Note that the call list of each function includes a pointer to the main KINSOL memory block, by which the function can access various data related to the KINSOL solution. The contents of this memory block are given in the file `kinsol_impl.h` (but not reproduced here, for the sake of space).

7.1 Initialization function

The type definition of `linit` is

`linit`

Definition `int (*linit)(KINMem kin_mem);`

Purpose The purpose of `linit` is to complete initializations for a specific linear solver, such as counters and statistics. It should also set pointers to data blocks that will later be passed to functions associated with the linear solver. The `linit` function is called once only, at the start of the problem, by `KINSol`.

Arguments `kin_mem` is the KINSOL memory pointer of type `KINMem`.

Return value An `linit` function should return 0 if it has successfully initialized the KINSOL linear solver, and a negative value otherwise.

7.2 Setup function

The type definition of `lsetup` is

`lsetup`

Definition `int (*lsetup)(KINMem kin_mem);`

Purpose The job of `lsetup` is to prepare the linear solver for subsequent calls to `lsolve`, in the solution of linear systems $Ax = b$. The exact nature of this system depends on the input `strategy` to `KINSol`. In the cases `strategy` = `KIN_NONE` or `KIN_LINESEARCH`, A is the Jacobian $J = \partial F/\partial u$. If `strategy` = `KIN_PICARD`, A is the approximate Jacobian matrix L. If `strategy` = `KIN_FP`, linear systems do not arise.

The `lsetup` function may call a user-supplied function, or a function within the linear solver module, to compute Jacobian-related data that is required by the linear solver. It may also preprocess that data as needed for `lsolve`, which may involve calling a generic function (such as for LU factorization). This data may be intended either for direct use (in a direct linear solver) or for use in a preconditioner (in a preconditioned iterative linear solver).

The `lsetup` function is not called at every Newton iteration, but only as frequently as the solver determines that it is appropriate to perform the setup task. In this way, Jacobian-related data generated by `lsetup` is expected to be used over a number of Newton iterations.

Arguments `kin_mem` is the KINSOL memory pointer of type `KINMem`.

Return value The `lsetup` function should return 0 if successful and a non-zero value otherwise.

Notes The current values of u and $F(u)$ can be accessed by `lsetup` through the fields `kin_uu` and `kin_fval` (respectively) in `kin_mem`. If needed, the scaling vectors `u_scale` and `f_scale` can be accessed by `lsetup` through the fields `kin_uscale` and `kin_fscale` (respectively) in `kin_mem`.

7.3 Solve function

The type definition of `lsolve` is

`lsolve`

Definition `int (*lsolve)(KINMem kin_mem, N_Vector x, N_Vector b,`
`realtype *sJpnorm, realtype *sFdotJp);`

Purpose The `lsolve` function must solve the linear system $Ax = b$, where A is either the Jacobian $J = \partial F/\partial u$ (evaluated at the current iterate), or the approximate Jacobian, L, in the case of Picard iteration. The right-hand side vector, b, is input.

Arguments

`kin_mem`	is the KINSOL memory pointer of type `KINMem`.
`x`	is a vector set to an initial guess prior to calling `lsolve`. On return it should contain the solution to $Jx = b$.
`b`	is the right-hand side vector b, set to $-F(u)$, evaluated at the current iterate.
`sJpnorm`	is a pointer to a real scalar to be computed by `lsolve`. The returned value `sJpnorm` should be equal to $\|D_F Jp\|_2$, the scaled L_2 norm of the product Jp, where p $(= x)$ is the computed solution of the linear system $Jp = b$, and the scaling is that given by D_F. This value is not needed in all cases. See below.
`sFdotJp`	is a pointer to a real scalar to be computed by `lsolve`. The returned value `sFdotJp` should be equal to $(D_F F) \cdot (D_F Jp)$, the dot product of the scaled F vector and the scaled vector Jp, where p $(= x)$ is the computed solution of the linear system $Jp = b$, and the scaling is that given by D_F. This value is not needed in all cases. See below.

Return value `lsolve` should return 0 if successful. If an error occurs and recovery could be possible by calling the `lsetup` function again, then it should return a positive value. Otherwise, `lsolve` should return a negative value.

Notes The current values of u and $F(u)$ can be accessed by `lsolve` through the fields `kin_uu` and `kin_fval` (respectively) in `kin_mem`, and the scaling vectors `u_scale` and `f_scale` can be accessed through the fields `kin_uscale` and `kin_fscale` (respectively) in `kin_mem`.

In the case of a direct solver, `sJpnorm` can be ignored, and `sFdotJp` can be computed with lines of the form

```
N_VProd(b, f_scale, b);
N_VProd(b, f_scale, b);
*sFdotJp = N_VDotProd(fval, b);
```

in which Jp is taken to be equal to the input right-hand side b, and `f_scale` and `fval` $(= F(u))$ are taken from `kin_mem`.

In the case of an iterative solver, the two terms, `sJpnorm` and `sFdotJp`, can be computed with lines of the form

```
ret = KINSpilsAtimes(kin_mem, x, b);
*sJpnorm = N_VWL2Norm(b, f_scale);
N_VProd(b, f_scale, b);
N_VProd(b, f_scale, b);
*sFdotJp = N_VDotProd(fval, b);
```

following the computation of the solution vector `x`, in which `f_scale` and `fval` $(= F(u))$ are taken from `kin_mem`.

The values `sFdotJp` and `sFdotJp` need not be set in all cases, and so for maximum efficiency, the `lsolve` function could do these calculations conditionally, depending on the value of the input `strategy` to `KINSol`, and the choice (given by `etachoice`) of Forcing Term in the Krylov iteration stopping test (see `KINSetEtaForm`). The precise conditions are as follows: First, if `strategy` is `KIN_FP`, neither of these quantities need to be computed. In the other cases, if the linear solver is iterative and `etachoice` = `KIN_ETACHOICE1` (the default) then both `sFdotJp` and `sFdotJp` must be set. If `strategy` is `KIN_LINESEARCH`, then `sFdotJp` must be set, regardless of the linear solver type.

The values of `strategy` and `etachoice` are available from the fields `kin_global_strategy` and `kin_etaflag` (respectively) in `kin_mem`.

7.4 Memory deallocation function

The type definition of `lfree` is

`lfree`

Definition	`void (*lfree)(KINMem kin_mem);`
Purpose	The `lfree` function should free any memory allocated by the linear solver.
Arguments	`kin_mem` is the KINSOL memory pointer of type `KINMem`.
Return value	The `lfree` function has no return value.
Notes	This function is called once a problem has been completed and the linear solver is no longer needed.

Chapter 8

General Use Linear Solver Components in SUNDIALS

In this chapter, we describe linear solver code components that are included in SUNDIALS, but which are of potential use as generic packages in themselves, either in conjunction with the use of SUNDIALS or separately.

These generic modules in SUNDIALS are organized in three families, the *dls* family, which includes direct linear solvers appropriate for sequential computations; the *sls* family, which includes sparse matrix solvers; and the *spils* family, which includes scaled preconditioned iterative (Krylov) linear solvers. The solvers in each family share common data structures and functions.

The *dls* family contains the following two generic linear solvers:

- The DENSE package, a linear solver for dense matrices either specified through a matrix type (defined below) or as simple arrays.
- The BAND package, a linear solver for banded matrices either specified through a matrix type (defined below) or as simple arrays.

Note that this family also includes the Blas/Lapack linear solvers (dense and band) available to the SUNDIALS solvers, but these are not discussed here.

The *sls* family contains a sparse matrix package and interfaces between it and two sparse direct solver packages:

- The KLU package, a linear solver for compressed-sparse-column matrices, [1, 7].
- The SUPERLUMT package, a threaded linear solver for compressed-sparse-column matrices, [2, 16, 9].

The *spils* family contains the following generic linear solvers:

- The SPGMR package, a solver for the scaled preconditioned GMRES method.
- The SPFGMR package, a solver for the scaled preconditioned Flexible GMRES method.
- The SPBCG package, a solver for the scaled preconditioned Bi-CGStab method.
- The SPTFQMR package, a solver for the scaled preconditioned TFQMR method.

For reasons related to installation, the names of the files involved in these packages begin with the prefix `sundials_`. But despite this, each of the *dls* and *spils* solvers is in fact generic, in that it is usable completely independently of SUNDIALS.

For the sake of space, the functions for the `dense` and `band` modules that work with a matrix type, and the functions in the SPGMR, SPFGMR, SPBCG, and SPTFQMR modules are only summarized briefly, since they are less likely to be of direct use in connection with a SUNDIALS solver. However, the functions for dense matrices treated as simple arrays and sparse matrices are fully described, because we expect that they will be useful in the implementation of preconditioners used with the combination of one of the SUNDIALS solvers and one of the *spils* linear solvers.

8.1 The DLS modules: DENSE and BAND

The files comprising the DENSE generic linear solver, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*/`include/sundials`)
  `sundials_direct.h`, `sundials_dense.h`,
  `sundials_types.h`, `sundials_math.h`, `sundials_config.h`
- source files (located in *srcdir*/`src/sundials`)
  `sundials_direct.c`, `sundials_dense.c`, `sundials_math.c`

The files comprising the BAND generic linear solver are as follows:

- header files (located in *srcdir*/`include/sundials`)
  `sundials_direct.h`, `sundials_band.h`,
  `sundials_types.h`, `sundials_math.h`, `sundials_config.h`
- source files (located in *srcdir*/`src/sundials`)
  `sundials_direct.c`, `sundials_band.c`, `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the DENSE and BAND packages by themselves.

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN` and `SUNMAX`, and the function `SUNRabs`.

The files listed above for either module can be extracted from the SUNDIALS *srcdir* and compiled by themselves into a separate library or into a larger user code.

8.1.1 Type DlsMat

The type `DlsMat`, defined in `sundials_direct.h` is a pointer to a structure defining a generic matrix, and is used with all linear solvers in the *dls* family:

```
typedef struct _DlsMat {
  int type;
  long int M;
  long int N;
  long int ldim;
  long int mu;
  long int ml;
  long int s_mu;
  realtype *data;
  long int ldata;
  realtype **cols;
} *DlsMat;
```

For the DENSE module, the relevant fields of this structure are as follows. Note that a dense matrix of type `DlsMat` need not be square.

type - `SUNDIALS_DENSE` (=1)

M - number of rows

N - number of columns

ldim - leading dimension (`ldim` $\geq$ `M`)

data - pointer to a contiguous block of `realtype` variables

ldata - length of the data array (= `ldim`·`N`). The (`i`,`j`)-th element of a dense matrix `A` of type `DlsMat` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) is given by the expression `(A->data)[0][j*M+i]`

cols - array of pointers. `cols[j]` points to the first element of the j-th column of the matrix in the array data. The (`i`,`j`)-th element of a dense matrix `A` of type `DlsMat` (with $0 \leq$ `i` $<$ `M` and $0 \leq$ `j` $<$ `N`) is given by the expression `(A->cols)[j][i]`

For the BAND module, the relevant fields of this structure are as follows (see Figure 8.1 for a diagram of the underlying data representation in a banded matrix of type `DlsMat`). Note that only square band matrices are allowed.

type - `SUNDIALS_BAND` (=2)

M - number of rows

N - number of columns (`N` = `M`)

mu - upper half-bandwidth, $0 \leq$ `mu` $<$ min(`M`,`N`)

ml - lower half-bandwidth, $0 \leq$ `ml` $<$ min(`M`,`N`)

s_mu - storage upper bandwidth, `mu` $\leq$ `s_mu` $<$ `N`. The LU decomposition routine writes the LU factors into the storage for A. The upper triangular factor U, however, may have an upper bandwidth as big as min(`N`-1,`mu`+`ml`) because of partial pivoting. The `s_mu` field holds the upper half-bandwidth allocated for A.

ldim - leading dimension (`ldim` $\geq$ `s_mu`)

data - pointer to a contiguous block of `realtype` variables. The elements of a banded matrix of type `DlsMat` are stored columnwise (i.e. columns are stored one on top of the other in memory). Only elements within the specified half-bandwidths are stored. `data` is a pointer to `ldata` contiguous locations which hold the elements within the band of A.

ldata - length of the data array (= `ldim`·(`s_mu`+`ml`+1)

cols - array of pointers. `cols[j]` is a pointer to the uppermost element within the band in the j-th column. This pointer may be treated as an array indexed from `s_mu`$-$`mu` (to access the uppermost element within the band in the j-th column) to `s_mu`+`ml` (to access the lowest element within the band in the j-th column). Indices from 0 to `s_mu`$-$`mu`-1 give access to extra storage elements required by the LU decomposition function. Finally, `cols[j][i-j+s_mu]` is the (i, j)-th element, $j-$`mu` $\leq i \leq j+$`ml`.

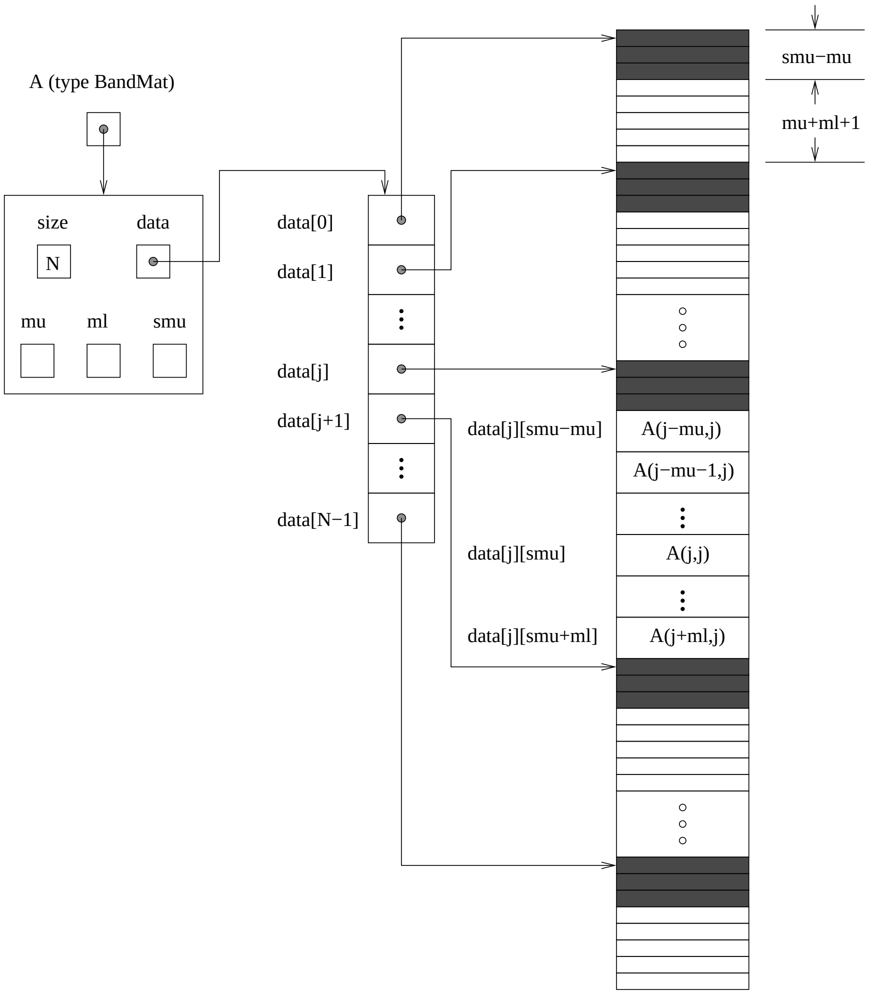


Figure 8.1: Diagram of the storage for a banded matrix of type `DlsMat`. Here `A` is an $N \times N$ band matrix of type `DlsMat` with upper and lower half-bandwidths `mu` and `ml`, respectively. The rows and columns of `A` are numbered from 0 to $N-1$ and the (i,j)-th element of `A` is denoted `A(i,j)`. The greyed out areas of the underlying component storage are used by the `BandGBTRF` and `BandGBTRS` routines.

8.1.2 Accessor macros for the DLS modules

The macros below allow a user to efficiently access individual matrix elements without writing out explicit data structure references and without knowing too much about the underlying element storage. The only storage assumption needed is that elements are stored columnwise and that a pointer to the j-th column of elements can be obtained via the DENSE_COL or BAND_COL macros. Users should use these macros whenever possible.

The following two macros are defined by the DENSE module to provide access to data in the DlsMat type:

- DENSE_ELEM

  Usage : `DENSE_ELEM(A,i,j) = a_ij;` or `a_ij = DENSE_ELEM(A,i,j);`

  DENSE_ELEM references the (i,j)-th element of the $M \times N$ DlsMat A, $0 \leq$ i $< M$, $0 \leq$ j $< N$.

- DENSE_COL

  Usage : `col_j = DENSE_COL(A,j);`

  DENSE_COL references the j-th column of the $M \times N$ DlsMat A, $0 \leq$ j $< N$. The type of the expression DENSE_COL(A,j) is realtype * . After the assignment in the usage above, col_j may be treated as an array indexed from 0 to $M - 1$. The (i, j)-th element of A is referenced by col_j[i].

The following three macros are defined by the BAND module to provide access to data in the DlsMat type:

- BAND_ELEM

  Usage : `BAND_ELEM(A,i,j) = a_ij;` or `a_ij = BAND_ELEM(A,i,j);`

  BAND_ELEM references the (i,j)-th element of the $N \times N$ band matrix A, where $0 \leq$ i, j $\leq N - 1$. The location (i,j) should further satisfy j$-$(A->mu) $\leq$ i $\leq$ j$+$(A->ml).

- BAND_COL

  Usage : `col_j = BAND_COL(A,j);`

  BAND_COL references the diagonal element of the j-th column of the $N \times N$ band matrix A, $0 \leq$ j $\leq N - 1$. The type of the expression BAND_COL(A,j) is realtype *. The pointer returned by the call BAND_COL(A,j) can be treated as an array which is indexed from $-$(A->mu) to (A->ml).

- BAND_COL_ELEM

  Usage : `BAND_COL_ELEM(col_j,i,j) = a_ij;` or `a_ij = BAND_COL_ELEM(col_j,i,j);`

  This macro references the (i,j)-th entry of the band matrix A when used in conjunction with BAND_COL to reference the j-th column through col_j. The index (i,j) should satisfy j$-$(A->mu) $\leq$ i $\leq$ j$+$(A->ml).

8.1.3 Functions in the DENSE module

The DENSE module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on dense matrices of type DlsMat. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for DlsMat dense matrices are available in the DENSE package. For full details, see the header files sundials_direct.h and sundials_dense.h.

- NewDenseMat: allocation of a DlsMat dense matrix;
- DestroyMat: free memory for a DlsMat matrix;

- `PrintMat`: print a `DlsMat` matrix to standard output.
- `NewLintArray`: allocation of an array of `long int` integers for use as pivots with `DenseGETRF` and `DenseGETRS`;
- `NewIntArray`: allocation of an array of `int` integers for use as pivots with the Lapack dense solvers;
- `NewRealArray`: allocation of an array of `realtype` for use as right-hand side with `DenseGETRS`;
- `DestroyArray`: free memory for an array;
- `SetToZero`: load a matrix with zeros;
- `AddIdentity`: increment a square matrix by the identity matrix;
- `DenseCopy`: copy one matrix to another;
- `DenseScale`: scale a matrix by a scalar;
- `DenseGETRF`: LU factorization with partial pivoting;
- `DenseGETRS`: solution of $Ax = b$ using LU factorization (for square matrices A);
- `DensePOTRF`: Cholesky factorization of a real symmetric positive matrix;
- `DensePOTRS`: solution of $Ax = b$ using the Cholesky factorization of A;
- `DenseGEQRF`: QR factorization of an $m \times n$ matrix, with $m \geq n$;
- `DenseORMQR`: compute the product $w = Qv$, with Q calculated using `DenseGEQRF`;
- `DenseMatvec`: compute the product $y = Ax$, for an M by N matrix A;

The following functions for small dense matrices are available in the DENSE package:

- `newDenseMat`

  `newDenseMat(m,n)` allocates storage for an `m` by `n` dense matrix. It returns a pointer to the newly allocated storage if successful. If the memory request cannot be satisfied, then `newDenseMat` returns `NULL`. The underlying type of the dense matrix returned is `realtype**`. If we allocate a dense matrix `realtype** a` by `a = newDenseMat(m,n)`, then `a[j][i]` references the (`i`,`j`)-th element of the matrix `a`, $0 \leq$ `i` $<$ `m`, $0 \leq$ `j` $< n$, and `a[j]` is a pointer to the first element in the `j`-th column of `a`. The location `a[0]` contains a pointer to `m` $\times$ `n` contiguous locations which contain the elements of `a`.

- `destroyMat`

  `destroyMat(a)` frees the dense matrix `a` allocated by `newDenseMat`;

- `newLintArray`

  `newLintArray(n)` allocates an array of `n` integers, all `long int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `newIntArray`

  `newIntArray(n)` allocates an array of `n` integers, all `int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `newRealArray`

  `newRealArray(n)` allocates an array of `n` `realtype` values. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `destroyArray`

  `destroyArray(p)` frees the array `p` allocated by `newLintArray`, `newIntArray`, or `newRealArray`;

- `denseCopy`

  `denseCopy(a,b,m,n)` copies the `m` by `n` dense matrix `a` into the `m` by `n` dense matrix `b`;

- `denseScale`

  `denseScale(c,a,m,n)` scales every element in the `m` by `n` dense matrix `a` by the scalar `c`;

- `denseAddIdentity`

  `denseAddIdentity(a,n)` increments the *square* `n` by `n` dense matrix `a` by the identity matrix I_n;

- `denseGETRF`

  `denseGETRF(a,m,n,p)` factors the `m` by `n` dense matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

  A successful LU factorization leaves the matrix `a` and the pivot array `p` with the following information:

  1. `p[k]` contains the row number of the pivot element chosen at the beginning of elimination step `k`, `k` $= 0, 1, ...,$`n`-1.
  2. If the unique LU factorization of `a` is given by $Pa = LU$, where P is a permutation matrix, L is an `m` by `n` lower trapezoidal matrix with all diagonal elements equal to 1, and U is an `n` by `n` upper triangular matrix, then the upper triangular part of `a` (including its diagonal) contains U and the strictly lower trapezoidal part of `a` contains the multipliers, $I - L$. If `a` is square, L is a unit lower triangular matrix.

     `denseGETRF` returns 0 if successful. Otherwise it encountered a zero diagonal element during the factorization, indicating that the matrix `a` does not have full column rank. In this case it returns the column index (numbered from one) at which it encountered the zero.

- `denseGETRS`

  `denseGETRS(a,n,p,b)` solves the `n` by `n` linear system $ax = b$. It assumes that `a` (of size `n` $\times$ `n`) has been LU-factored and the pivot array `p` has been set by a successful call to `denseGETRF(a,n,n,p)`. The solution x is written into the `b` array.

- `densePOTRF`

  `densePOTRF(a,m)` calculates the Cholesky decomposition of the `m` by `m` dense matrix `a`, assumed to be symmetric positive definite. Only the lower triangle of `a` is accessed and overwritten with the Cholesky factor.

- `densePOTRS`

  `densePOTRS(a,m,b)` solves the `m` by `m` linear system $ax = b$. It assumes that the Cholesky factorization of `a` has been calculated in the lower triangular part of `a` by a successful call to `densePOTRF(a,m)`.

- `denseGEQRF`

  `denseGEQRF(a,m,n,beta,wrk)` calculates the QR decomposition of the `m` by `n` matrix `a` (`m` $\geq$ `n`) using Householder reflections. On exit, the elements on and above the diagonal of `a` contain the `n` by `n` upper triangular matrix `R`; the elements below the diagonal, with the array `beta`, represent the orthogonal matrix `Q` as a product of elementary reflectors. The real array `wrk`, of length `m`, must be provided as temporary workspace.

- `denseORMQR`

  `denseORMQR(a,m,n,beta,v,w,wrk)` calculates the product $w = Qv$ for a given vector `v` of length `n`, where the orthogonal matrix Q is encoded in the `m` by `n` matrix `a` and the vector `beta` of length `n`, after a successful call to `denseGEQRF(a,m,n,beta,wrk)`. The real array `wrk`, of length `m`, must be provided as temporary workspace.

- `denseMatvec`

  `denseMatvec(a,x,y,m,n)` calculates the product $y = ax$ for a given vector `x` of length `n`, and `m` by `n` matrix `a`.

8.1.4 Functions in the BAND module

The BAND module defines two sets of functions with corresponding names. The first set contains functions (with names starting with a capital letter) that act on band matrices of type `DlsMat`. The second set contains functions (with names starting with a lower case letter) that act on matrices represented as simple arrays.

The following functions for `DlsMat` banded matrices are available in the BAND package. For full details, see the header files `sundials_direct.h` and `sundials_band.h`.

- `NewBandMat`: allocation of a `DlsMat` band matrix;
- `DestroyMat`: free memory for a `DlsMat` matrix;
- `PrintMat`: print a `DlsMat` matrix to standard output.
- `NewLintArray`: allocation of an array of `int` integers for use as pivots with `BandGBRF` and `BandGBRS`;
- `NewIntArray`: allocation of an array of `int` integers for use as pivots with the Lapack band solvers;
- `NewRealArray`: allocation of an array of `realtype` for use as right-hand side with `BandGBRS`;
- `DestroyArray`: free memory for an array;
- `SetToZero`: load a matrix with zeros;
- `AddIdentity`: increment a square matrix by the identity matrix;
- `BandCopy`: copy one matrix to another;
- `BandScale`: scale a matrix by a scalar;
- `BandGBTRF`: LU factorization with partial pivoting;
- `BandGBTRS`: solution of $Ax = b$ using LU factorization;
- `BandMatvec`: compute the product $y = Ax$, for a square band matrix A;

The following functions for small band matrices are available in the BAND package:

- `newBandMat`

  `newBandMat(n, smu, ml)` allocates storage for an `n` by `n` band matrix with lower half-bandwidth `ml`.

- `destroyMat`

  `destroyMat(a)` frees the band matrix `a` allocated by `newBandMat`;

- `newLintArray`

  `newLintArray(n)` allocates an array of `n` integers, all `long int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `newIntArray`

  `newIntArray(n)` allocates an array of `n` integers, all `int`. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `newRealArray`

  `newRealArray(n)` allocates an array of `n realtype` values. It returns a pointer to the first element in the array if successful. It returns `NULL` if the memory request could not be satisfied.

- `destroyArray`

  `destroyArray(p)` frees the array `p` allocated by `newLintArray`, `newIntArray`, or `newRealArray`;

- `bandCopy`

  `bandCopy(a,b,n,a_smu, b_smu,copymu, copyml)` copies the `n` by `n` band matrix `a` into the `n` by `n` band matrix `b`;

- `bandScale`

  `bandScale(c,a,n,mu,ml,smu)` scales every element in the `n` by `n` band matrix `a` by `c`;

- `bandAddIdentity`

  `bandAddIdentity(a,n,smu)` increments the `n` by `n` band matrix `a` by the identity matrix;

- `bandGETRF`

  `bandGETRF(a,n,mu,ml,smu,p)` factors the `n` by `n` band matrix `a`, using Gaussian elimination with row pivoting. It overwrites the elements of `a` with its LU factors and keeps track of the pivot rows chosen in the pivot array `p`.

- `bandGETRS`

  `bandGETRS(a,n,smu,ml,p,b)` solves the `n` by `n` linear system $ax = b$. It assumes that `a` (of size `n` $\times$ `n`) has been LU-factored and the pivot array `p` has been set by a successful call to `bandGETRF(a,n,mu,ml,smu,p)`. The solution x is written into the `b` array.

- `bandMatvec`

  `bandMatvec(a,x,y,n,mu,ml,smu)` calculates the product $y = ax$ for a given vector `x` of length `n`, and `n` by `n` band matrix `a`.

8.2 The SLS module

SUNDIALS provides a compressed-sparse-column matrix type and sparse matrix support functions. In addition, SUNDIALS provides interfaces to the publically available KLU and SuperLU_MT sparse direct solver packages. The files comprising the SLS matrix module, used in the KLU and SUPERLUMT linear solver packages, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_sparse.h`, `sundials_klu_impl.h`,
  `sundials_superlumt_impl.h`, `sundials_types.h`,
  `sundials_math.h`, `sundials_config.h`

- source files (located in *srcdir*`/src/sundials`)
  `sundials_sparse.c`, `sundials_math.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are relevant to the SLS package by itself:

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  `#define SUNDIALS_DOUBLE_PRECISION 1`
  `#define SUNDIALS_SINGLE_PRECISION 1`
  `#define SUNDIALS_EXTENDED_PRECISION 1`
- (optional) use of generic math functions: `#define SUNDIALS_USE_GENERIC_MATH 1`

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN` and `SUNMAX`, and the function `SUNRabs`.

8.2.1 Type SlsMat

The type `SlsMat`, defined in `sundials_sparse.h` is a pointer to a structure defining a generic compressed-sparse-column matrix, and is used with all linear solvers in the *sls* family:

```
typedef struct _SlsMat {
  int M;
  int N;
  int NNZ;
  realtype *data;
  int *rowvals;
  int *colptrs;
} *SlsMat;
```

The fields of this structure are as follows (see Figure 8.2 for a diagram of the underlying compressed-sparse-column representation in a sparse matrix of type `SlsMat`). Note that a sparse matrix of type `SlsMat` need not be square.

M - number of rows

N - number of columns

NNZ - maximum number of nonzero entries in the matrix (allocated length of `data` and `rowvals` arrays)

data - pointer to a contiguous block of `realtype` variables (of length `NNZ`), containing the values of the nonzero entries in the matrix

rowvals - pointer to a contiguous block of `int` variables (of length `NNZ`), containing the row indices of each nonzero entry held in `data`

colptrs - pointer to a contiguous block of `int` variables (of length `N+1`). Each entry provides the index of the first column entry into the `data` and `rowvals` arrays, e.g. if `colptr[3]=7`, then the first nonzero entry in the fourth column of the matrix is located in `data[7]`, and is located in row `rowvals[7]` of the matrix. The last entry contains the total number of nonzero values in the matrix and hence points one past the end of the active data in the `data` and `rowvals` arrays.

For example, the 5×4 matrix

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 3 & 0 & 0 & 2 \\ 0 & 7 & 0 & 0 \\ 1 & 0 & 0 & 9 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

could be stored in a `SlsMat` structure as either

```
M = 5;
N = 4;
NNZ = 8;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0};
rowvals = {1, 3, 0, 2, 0, 1, 3, 4};
colptrs = {0, 2, 4, 5, 8};
```

or

```
M = 5;
N = 4;
NNZ = 10;
data = {3.0, 1.0, 3.0, 7.0, 1.0, 2.0, 9.0, 5.0, *, *};
rowvals = {1, 3, 0, 2, 0, 1, 3, 4, *, *};
colptrs = {0, 2, 4, 5, 8};
```

where the first has no unused space, and the second has additional storage (the entries marked with * may contain any values). Note in both cases that the final value in `colptrs` is 8. The work associated with operations on the sparse matrix is proportional to this value and so one should use the best understanding of the number of nonzeroes here.

8.2.2 Functions in the SLS module

The SLS module defines functions that act on sparse matrices of type `SlsMat`. For full details, see the header file `sundials_sparse.h`.

- `NewSparseMat`

  `NewSparseMat(M, N, NNZ)` allocates storage for an `M` by `N` sparse matrix, with storage for up to `NNZ` nonzero entries.

- `SlsConvertDls`

  `SlsConvertDls(A)` converts a dense or band matrix `A` of type `DlsMat` into a new sparse matrix of type `SlsMat` by retaining only the nonzero values of the matrix `A`.

- `DestroySparseMat`

  `DestroySparseMat(A)` frees the memory for a sparse matrix `A` allocated by either `NewSparseMat` or `SlsConvertDls`.

- `SlsSetToZero(A)` zeros out the `SlsMat` matrix `A`. The storage for `A` is left unchanged.

- `CopySparseMat`

  `CopySparseMat(A, B)` copies the `SlsMat` `A` into the `SlsMat` `B`. It is assumed that the matrices have the same row/column dimensions. If `B` has insufficient storage to hold all the nonzero entries of `A`, the data and row index arrays in `B` are reallocated to match those in `A`.

- `ScaleSparseMat`

  `ScaleSparseMat(c, A)` scales every element in the `SlsMat` `A` by the `realtype` scalar `c`.

- `AddIdentitySparseMat`

  `AddIdentitySparseMat(A)` increments the `SlsMat` `A` by the identity matrix. If `A` is not square, only the existing diagonal values are incremented. Resizes the `data` and `rowvals` arrays of `A` to allow for new nonzero entries on the diagonal.

- `SlsAddMat`

  `SlsAddMat(A, B)` adds two `SlsMat` matrices `A` and `B`, placing the result back in `A`. Resizes the `data` and `rowvals` arrays of `A` upon completion to exactly match the nonzero storage for the result. Upon successful completion, the return value is zero; otherwise -1 is returned.

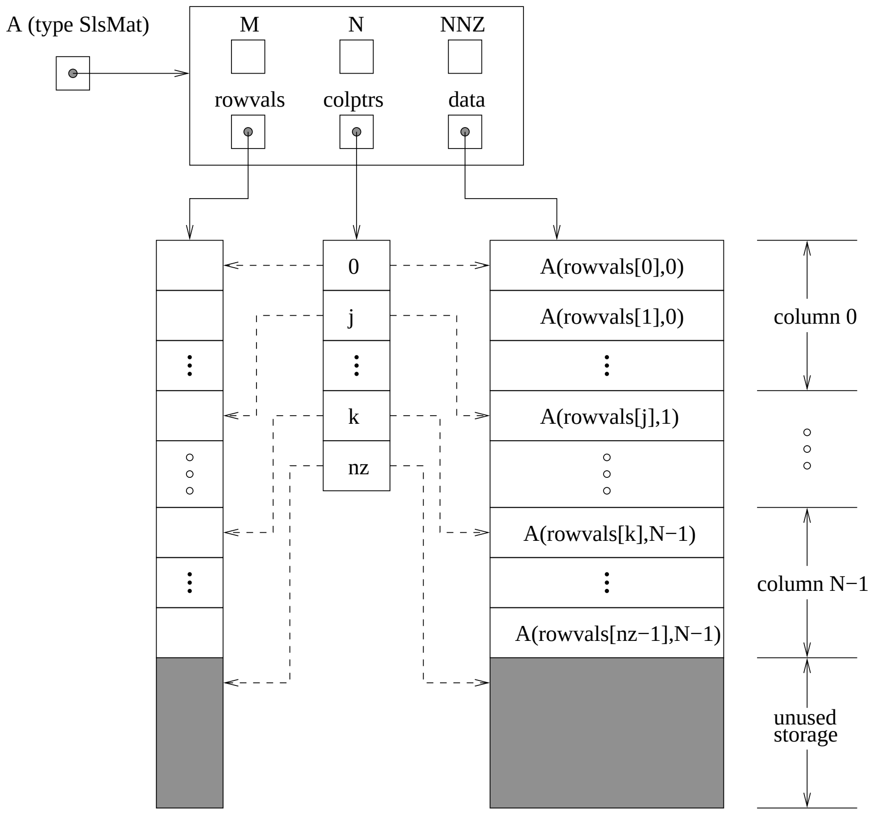


Figure 8.2: Diagram of the storage for a compressed-sparse-column matrix of type `SlsMat`. Here `A` is an $M \times N$ sparse matrix of type `SlsMat` with storage for up to `NNZ` nonzero entries (the allocated length of both `data` and `rowvals`). The entries in `rowvals` may assume values from 0 to $M - 1$, corresponding to the row index (zero-based) of each nonzero value. The entries in `data` contain the values of the nonzero entries, with the row i, column j entry of `A` (again, zero-based) denoted as `A(i,j)`. The `colptrs` array contains $N + 1$ entries; the first N denote the starting index of each column within the `rowvals` and `data` arrays, while the final entry points one past the final nonzero entry. Here, although `NNZ` values are allocated, only `nz` are actually filled in; the greyed-out portions of `data` and `rowvals` indicate extra allocated space.

- `ReallocSparseMat`

  `ReallocSparseMat(A)` eliminates unused storage in the `SlsMat A` by resizing the internal `data` and `rowvals` arrays to contain exactly `colptrs[N]` values.

- `SlsMatvec`

  `SlsMatvec(A, x, y)` computes the sparse matrix-vector product, $y = Ax$. If the `SlsMat A` is a sparse matrix of dimension $M \times N$, then it is assumed that `x` is a `realtype` array of length N, and `y` is a `realtype` array of length M. Upon successful completion, the return value is zero; otherwise -1 is returned.

- `PrintSparseMat`

  `PrintSparseMat(A)` Prints the `SlsMat` matrix `A` to standard output.

8.2.3 The KLU solver

KLU is a sparse matrix factorization and solver library written by Tim Davis [1, 7]. KLU has a symbolic factorization routine that computes the permutation of the linear system matrix to block triangular form and the permutations that will pre-order the diagonal blocks (the only ones that need to be factored) to reduce fill-in (using AMD, COLAMD, CHOLAMD, natural, or an ordering given by the user). Note that SUNDIALS uses the COLAMD ordering by default with KLU.

KLU breaks the factorization into two separate parts. The first is a symbolic factorization and the second is a numeric factorization that returns the factored matrix along with final pivot information. KLU also has a refactor routine that can be called instead of the numeric factorization. This routine will reuse the pivot information. This routine also returns diagnostic information that a user can examine to determine if numerical stability is being lost and a full numerical factorization should be done instead of the refactor.

The KLU interface in SUNDIALS will perform the symbolic factorization once. It then calls the numerical factorization once and will call the refactor routine until estimates of the numerical conditioning suggest a new factorization should be completed. The KLU interface also has a `ReInit` routine that can be used to force a full refactorization at the next solver setup call.

In order to use the SUNDIALS interface to KLU, it is assumed that KLU has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with KLU (see Appendix A for details).

Designed for serial calculations only, KLU is supported for calculations employing SUNDIALS' serial or shared-memory parallel NVECTOR modules (see Sections 6.1, 6.3 and 6.4 for details).

8.2.4 The SUPERLUMT solver

SUPERLUMT is a threaded sparse matrix factorization and solver library written by X. Sherry Li [2, 16, 9]. The package performs matrix factorization using threads to enhance efficiency in shared memory parallel environments. It should be noted that threads are only used in the factorization step.

In order to use the SUNDIALS interface to SUPERLUMT, it is assumed that SUPERLUMT has been installed on the system prior to installation of SUNDIALS, and that SUNDIALS has been configured appropriately to link with SUPERLUMT (see Appendix A for details).

Designed for serial and threaded calculations only, SUPERLUMT is supported for calculations employing SUNDIALS' serial or shared-memory parallel NVECTOR modules (see Sections 6.1, 6.3 and 6.4 for details).

8.3 The SPILS modules: SPGMR, SPFGMR, SPBCG, and SPTFQMR

The *spils* modules contain implementations of some of the most commonly use scaled preconditioned Krylov solvers. A linear solver module from the *spils* family can be used in conjunction with any

NVECTOR implementation library.

8.3.1 The SPGMR module

The SPGMR package, in the files `sundials_spgmr.h` and `sundials_spgmr.c`, includes an implementation of the scaled preconditioned GMRES method. A separate code module, implemented in `sundials_iterative.(h,c)`, contains auxiliary functions that support SPGMR, as well as the other Krylov solvers in SUNDIALS (SPFGMR, SPBCG, and SPTFQMR). For full details, including usage instructions, see the header files `sundials_spgmr.h` and `sundials_iterative.h`.

The files comprising the SPGMR generic linear solver, and their locations in the SUNDIALS *srcdir*, are as follows:

- header files (located in *srcdir*`/include/sundials`)
  `sundials_spgmr.h`, `sundials_iterative.h`, `sundials_nvector.h`,
  `sundials_types.h`, `sundials_math.h`, `sundials_config.h`
- source files (located in *srcdir*`/src/sundials`)
  `sundials_spgmr.c`, `sundials_iterative.c`, `sundials_nvector.c`

Only two of the preprocessing directives in the header file `sundials_config.h` are required to use the SPGMR package by itself:

- (required) definition of the precision of the SUNDIALS type `realtype`. One of the following lines must be present:
  ```
  #define SUNDIALS_DOUBLE_PRECISION 1
  #define SUNDIALS_SINGLE_PRECISION 1
  #define SUNDIALS_EXTENDED_PRECISION 1
  ```
- (optional) use of generic math functions:
  ```
  #define SUNDIALS_USE_GENERIC_MATH 1
  ```

The `sundials_types.h` header file defines the SUNDIALS `realtype` and `booleantype` types and the macro `RCONST`, while the `sundials_math.h` header file is needed for the macros `SUNMIN`, `SUNMAX`, and `SUNSQR`, and the functions `SUNRabs` and `SUNRsqrt`.

The generic NVECTOR files, `sundials_nvector.(h,c)` are needed for the definition of the generic `N_Vector` type and functions. The NVECTOR functions used by the SPGMR module are: `N_VDotProd`, `N_VLinearSum`, `N_VScale`, `N_VProd`, `N_VDiv`, `N_VConst`, `N_VClone`, `N_VCloneVectorArray`, `N_VDestroy`, and `N_VDestroyVectorArray`.

The nine files listed above can be extracted from the SUNDIALS *srcdir* and compiled by themselves into an SPGMR library or into a larger user code.

The following functions are available in the SPGMR package:

- `SpgmrMalloc`: allocation of memory for `SpgmrSolve`;
- `SpgmrSolve`: solution of $Ax = b$ by the SPGMR method;
- `SpgmrFree`: free memory allocated by `SpgmrMalloc`.

The following functions are available in the support package `sundials_iterative.(h,c)`:

- `ModifiedGS`: performs modified Gram-Schmidt procedure;
- `ClassicalGS`: performs classical Gram-Schmidt procedure;
- `QRfact`: performs QR factorization of Hessenberg matrix;
- `QRsol`: solves a least squares problem with a Hessenberg matrix factored by `QRfact`.

8.3.2 The SPFGMR module

The SPFGMR package, in the files `sundials_spfgmr.h` and `sundials_spfgmr.c`, includes an implementation of the scaled preconditioned Flexible GMRES method. For full details, including usage instructions, see the file `sundials_spfgmr.h`.

The files needed to use the SPFGMR module by itself are the same as for the SPGMR module, but with `sundials_spfgmr.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPFGMR package:

- `SpfgmrMalloc`: allocation of memory for `SpfgmrSolve`;
- `SpfgmrSolve`: solution of $Ax = b$ by the SPFGMR method;
- `SpfgmrFree`: free memory allocated by `SpfgmrMalloc`.

8.3.3 The SPBCG module

The SPBCG package, in the files `sundials_spbcgs.h` and `sundials_spbcgs.c`, includes an implementation of the scaled preconditioned Bi-CGStab method. For full details, including usage instructions, see the file `sundials_spbcgs.h`.

The files needed to use the SPBCG module by itself are the same as for the SPGMR module, but with `sundials_spbcgs.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPBCG package:

- `SpbcgMalloc`: allocation of memory for `SpbcgSolve`;
- `SpbcgSolve`: solution of $Ax = b$ by the SPBCG method;
- `SpbcgFree`: free memory allocated by `SpbcgMalloc`.

8.3.4 The SPTFQMR module

The SPTFQMR package, in the files `sundials_sptfqmr.h` and `sundials_sptfqmr.c`, includes an implementation of the scaled preconditioned TFQMR method. For full details, including usage instructions, see the file `sundials_sptfqmr.h`.

The files needed to use the SPTFQMR module by itself are the same as for the SPGMR module, but with `sundials_sptfqmr.(h,c)` in place of `sundials_spgmr.(h,c)`.

The following functions are available in the SPTFQMR package:

- `SptfqmrMalloc`: allocation of memory for `SptfqmrSolve`;
- `SptfqmrSolve`: solution of $Ax = b$ by the SPTFQMR method;
- `SptfqmrFree`: free memory allocated by `SptfqmrMalloc`.

Appendix A

SUNDIALS Package Installation Procedure

The installation of any SUNDIALS package is accomplished by installing the SUNDIALS suite as a whole, according to the instructions that follow. The same procedure applies whether or not the downloaded file contains one or all solvers in SUNDIALS.

The SUNDIALS suite (or individual solvers) are distributed as compressed archives (`.tar.gz`). The name of the distribution archive is of the form *solver*-`x.y.z.tar.gz`, where *solver* is one of: `sundials`, `cvode`, `cvodes`, `arkode`, `ida`, `idas`, or `kinsol`, and `x.y.z` represents the version number (of the SUNDIALS suite or of the individual solver) . To begin the installation, first uncompress and expand the sources, by issuing

```
% tar xzf solver-x.y.z.tar.gz
```

This will extract source files under a directory *solver*-`x.y.z`.

Starting with version `2.6.0` of SUNDIALS, CMake is the only supported method of installation. The explanations on the installation procedure begins with a few common observations:

- The remainder of this chapter will follow these conventions:

  srcdir is the directory *solver*-`x.y.z` created above; i.e., the directory containing the SUNDIALS sources.

  builddir is the (temporary) directory under which SUNDIALS is built.

  instdir is the directory under which the SUNDIALS exported header files and libraries will be installed. Typically, header files are exported under a directory *instdir*`/include` while libraries are installed under *instdir*`/lib`, with *instdir* specified at configuration time.

- For SUNDIALS CMake-based installation, in-source builds are prohibited; in other words, the build directory *builddir* can **not** be the same as *srcdir* and such an attempt will lead to an error. This prevents "polluting" the source tree and allows efficient builds for different configurations and/or options.

- The installation directory *instdir* can **not** be the same as the source directory *srcdir*.

- By default, only the libraries and header files are exported to the installation directory *instdir*. If enabled by the user (with the appropriate toggle for CMake), the examples distributed with SUNDIALS will be built together with the solver libraries but the installation step will result in exporting (by default in a subdirectory of the installation directory) the example sources and sample outputs together with automatically generated configuration files that reference the *installed* SUNDIALS headers and libraries. As such, these configuration files for the SUNDIALS examples can be used as "templates" for your own problems. CMake installs `CMakeLists.txt` files and also (as an option available only under Unix/Linux) `Makefile` files. Note this installation approach also allows the option of building the SUNDIALS examples without having to install them. (This can be used as a sanity check for the freshly built libraries.)

- Even if generation of shared libraries is enabled, only static libraries are created for the FCMIX modules. (Because of the use of fixed names for the Fortran user-provided subroutines, FCMIX shared libraries would result in "undefined symbol" errors at link time.)

A.1 CMake-based installation

CMake-based installation provides a platform-independent build system. CMake can generate Unix and Linux Makefiles, as well as KDevelop, Visual Studio, and (Apple) XCode project files from the same configuration file. In addition, CMake also provides a GUI front end and which allows an interactive build and installation process.

The SUNDIALS build process requires CMake version `2.8.1` or higher and a working compiler. On Unix-like operating systems, it also requires Make (and `curses`, including its development libraries, for the GUI front end to CMake, `ccmake`), while on Windows it requires Visual Studio. While many Linux distributions offer CMake, the version included is probably out of date. Many new CMake features have been added recently, and you should download the latest version from `http://www.cmake.org`. Build instructions for CMake (only necessary for Unix-like systems) can be found on the CMake website. Once CMake is installed, Linux/Unix users will be able to use `ccmake`, while Windows users will be able to use `CMakeSetup`.

As previously noted, when using CMake to configure, build and install SUNDIALS, it is always required to use a separate build directory. While in-source builds are possible, they are explicitly prohibited by the SUNDIALS CMake scripts (one of the reasons being that, unlike autotools, CMake does not provide a `make distclean` procedure and it is therefore difficult to clean-up the source tree after an in-source build). By ensuring a separate build directory, it is an easy task for the user to clean-up all traces of the build by simply removing the build directory. CMake does generate a `make clean` which will remove files generated by the compiler and linker.

A.1.1 Configuring, building, and installing on Unix-like systems

The default CMake configuration will build all included solvers and associated examples and will build static and shared libraries. The *installdir* defaults to `/usr/local` and can be changed by setting the `CMAKE_INSTALL_PREFIX` variable. Support for FORTRAN and all other options are disabled.

CMake can be used from the command line with the `cmake` command, or from a `curses`-based GUI by using the `ccmake` command. Examples for using both methods will be presented. For the examples shown it is assumed that there is a top level SUNDIALS directory with appropriate source, build and install directories:

```
% mkdir (...)sundials/instdir
% mkdir (...)sundials/builddir
% cd (...)sundials/builddir
```

Building with the GUI

Using CMake with the GUI follows this general process:

- Select and modify values, run configure (`c` key)
- New values are denoted with an asterisk
- To set a variable, move the cursor to the variable and press enter
  - If it is a boolean (ON/OFF) it will toggle the value
  - If it is string or file, it will allow editing of the string

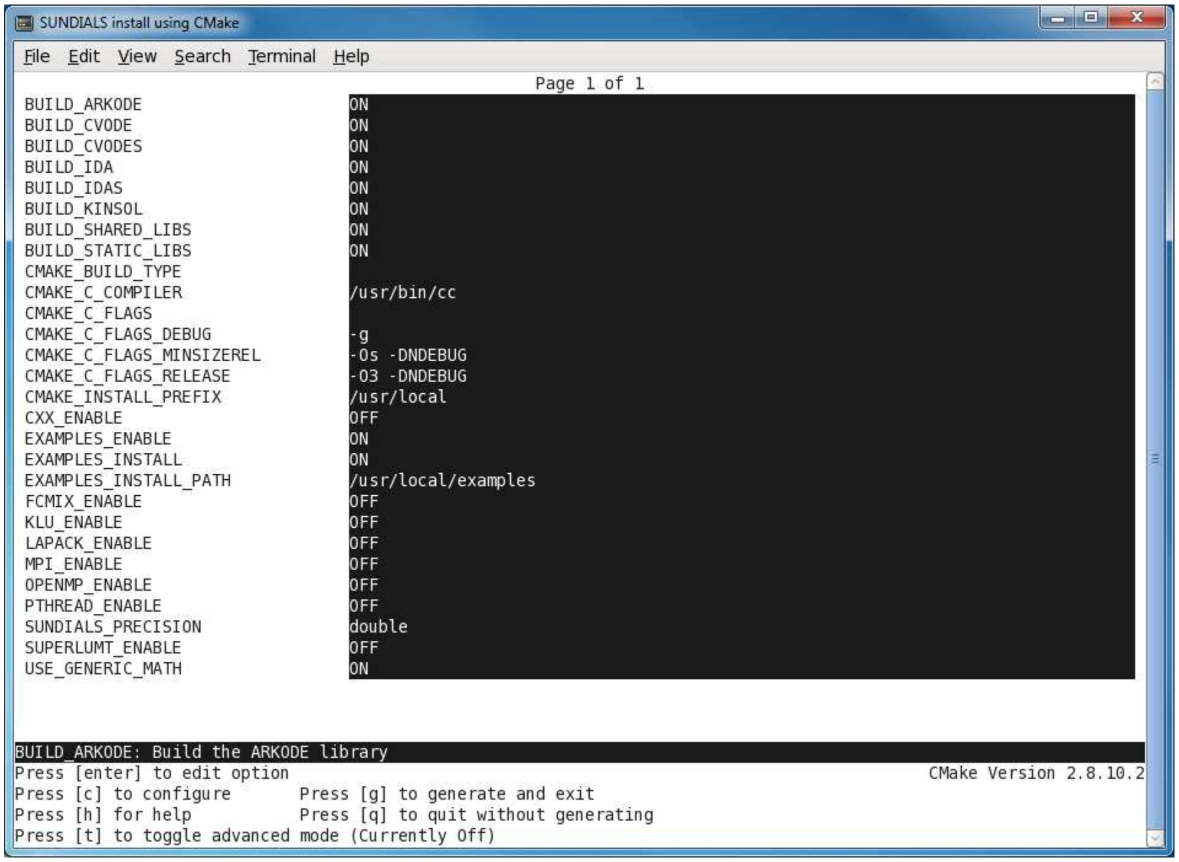
  – For file and directories, the `<tab>` key can be used to complete

- Repeat until all values are set as desired and the generate option is available (`g` key)
- Some variables (advanced variables) are not visible right away
- To see advanced variables, toggle to advanced mode (`t` key)
- To search for a variable press `/` key, and to repeat the search, press the `n` key

To build the default configuration using the GUI, from the *builddir* enter the ccmake command and point to the *srcdir*:

```
% ccmake ../srcdir
```

The default configuration screen is shown in Figure A.1.


SUNDIALS install using CMake
File Edit View Search Terminal Help

```
                                     Page 1 of 1
BUILD_ARKODE                     ON
BUILD_CVODE                      ON
BUILD_CVODES                     ON
BUILD_IDA                        ON
BUILD_IDAS                       ON
BUILD_KINSOL                     ON
BUILD_SHARED_LIBS                ON
BUILD_STATIC_LIBS                ON
CMAKE_BUILD_TYPE
CMAKE_C_COMPILER                 /usr/bin/cc
CMAKE_C_FLAGS
CMAKE_C_FLAGS_DEBUG              -g
CMAKE_C_FLAGS_MINSIZEREL         -Os -DNDEBUG
CMAKE_C_FLAGS_RELEASE            -O3 -DNDEBUG
CMAKE_INSTALL_PREFIX             /usr/local
CXX_ENABLE                       OFF
EXAMPLES_ENABLE                  ON
EXAMPLES_INSTALL                 ON
EXAMPLES_INSTALL_PATH            /usr/local/examples
FCMIX_ENABLE                     OFF
KLU_ENABLE                       OFF
LAPACK_ENABLE                    OFF
MPI_ENABLE                       OFF
OPENMP_ENABLE                    OFF
PTHREAD_ENABLE                   OFF
SUNDIALS_PRECISION               double
SUPERLUMT_ENABLE                 OFF
USE_GENERIC_MATH                 ON


BUILD_ARKODE: Build the ARKODE library
Press [enter] to edit option                                               CMake Version 2.8.10.2
Press [c] to configure       Press [g] to generate and exit
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure A.1: Default configuration screen. Note: Initial screen is empty. To get this default configuration, press 'c' repeatedly (accepting default values denoted with asterisk) until the 'g' option is available.

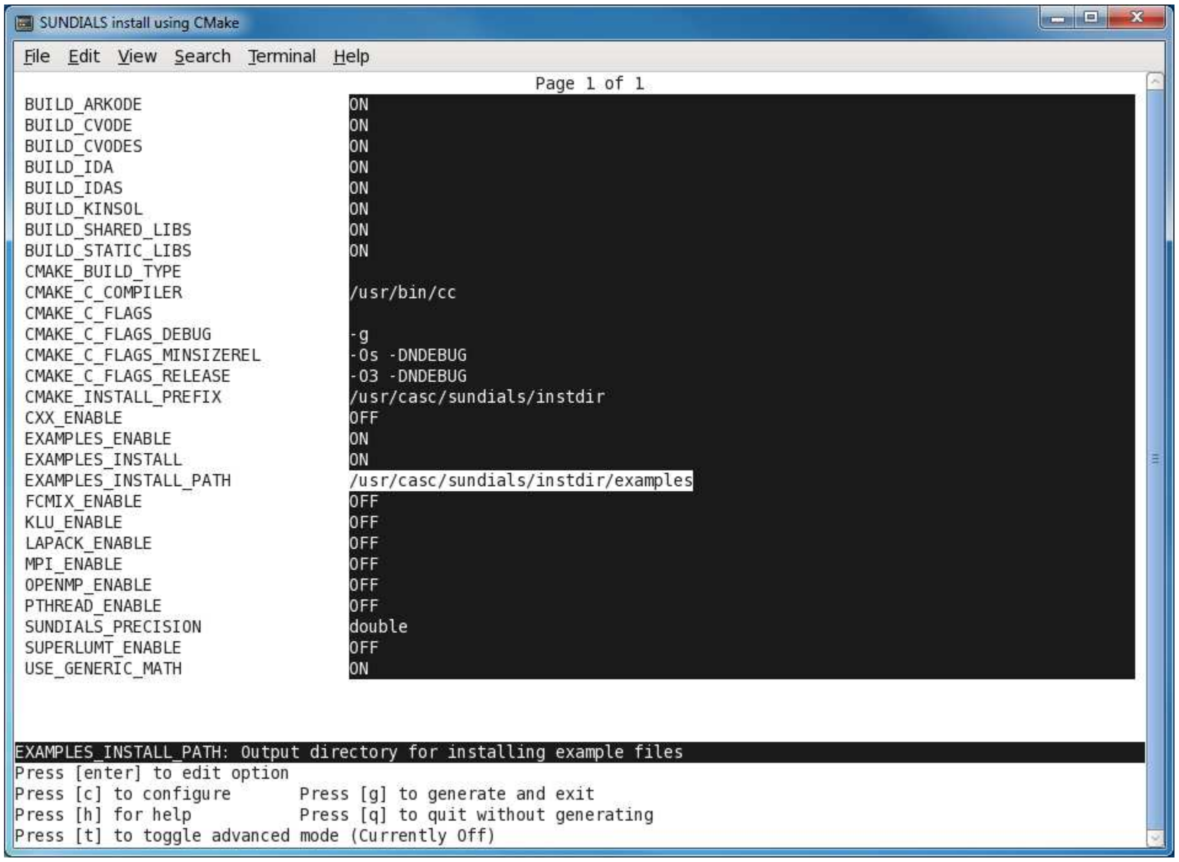
The default *instdir* for both SUNDIALS and corresponding examples can be changed by setting the `CMAKE_INSTALL_PREFIX` and the `EXAMPLES_INSTALL_PATH` as shown in figure A.2.

Pressing the (`g` key) will generate makefiles including all dependencies and all rules to build SUNDIALS on this system. Back at the command prompt, you can now run:

```
% make
```

To install SUNDIALS in the installation directory specified in the configuration, simply run:

```
% make install
```


SUNDIALS install using CMake
File Edit View Search Terminal Help

```
                                   Page 1 of 1
BUILD_ARKODE                     ON
BUILD_CVODE                      ON
BUILD_CVODES                     ON
BUILD_IDA                        ON
BUILD_IDAS                       ON
BUILD_KINSOL                     ON
BUILD_SHARED_LIBS                ON
BUILD_STATIC_LIBS                ON
CMAKE_BUILD_TYPE
CMAKE_C_COMPILER                 /usr/bin/cc
CMAKE_C_FLAGS
CMAKE_C_FLAGS_DEBUG              -g
CMAKE_C_FLAGS_MINSIZEREL         -Os -DNDEBUG
CMAKE_C_FLAGS_RELEASE            -O3 -DNDEBUG
CMAKE_INSTALL_PREFIX             /usr/casc/sundials/instdir
CXX_ENABLE                       OFF
EXAMPLES_ENABLE                  ON
EXAMPLES_INSTALL                 ON
EXAMPLES_INSTALL_PATH            /usr/casc/sundials/instdir/examples
FCMIX_ENABLE                     OFF
KLU_ENABLE                       OFF
LAPACK_ENABLE                    OFF
MPI_ENABLE                       OFF
OPENMP_ENABLE                    OFF
PTHREAD_ENABLE                   OFF
SUNDIALS_PRECISION               double
SUPERLUMT_ENABLE                 OFF
USE_GENERIC_MATH                 ON


EXAMPLES_INSTALL_PATH: Output directory for installing example files
Press [enter] to edit option
Press [c] to configure       Press [g] to generate and exit
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure A.2: Changing the *instdir* for SUNDIALS and corresponding examples

Building from the command line

Using CMake from the command line is simply a matter of specifying CMake variable settings with the cmake command. The following will build the default configuration:

```
% cmake -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> ../srcdir
% make
% make install
```

A.1.2 Configuration options (Unix/Linux)

A complete list of all available options for a CMake-based SUNDIALS configuration is provide below. Note that the default values shown are for a typical configuration on a Linux system and are provided as illustration only.

BUILD_ARKODE - Build the ARKODE library
Default: ON

BUILD_CVODE - Build the CVODE library
Default: ON

BUILD_CVODES - Build the CVODES library
Default: ON

`BUILD_IDA` - Build the IDA library
Default: ON

`BUILD_IDAS` - Build the IDAS library
Default: ON

`BUILD_KINSOL` - Build the KINSOL library
Default: ON

`BUILD_SHARED_LIBS` - Build shared libraries
Default: OFF

`BUILD_STATIC_LIBS` - Build static libraries
Default: ON

`CMAKE_BUILD_TYPE` - Choose the type of build, options are: None (CMAKE_C_FLAGS used) Debug Release RelWithDebInfo MinSizeRel
Default:

`CMAKE_C_COMPILER` - C compiler
Default: /usr/bin/cc

`CMAKE_C_FLAGS` - Flags for C compiler
Default:

`CMAKE_C_FLAGS_DEBUG` - Flags used by the compiler during debug builds
Default: -g

`CMAKE_C_FLAGS_MINSIZEREL` - Flags used by the compiler during release minsize builds
Default: -Os -DNDEBUG

`CMAKE_C_FLAGS_RELEASE` - Flags used by the compiler during release builds
Default: -O3 -DNDEBUG

`CMAKE_Fortran_COMPILER` - Fortran compiler
Default: /usr/bin/gfortran
Note: Fortran support (and all related options) are triggered only if either Fortran-C support is enabled (`FCMIX_ENABLE` is ON) or Blas/Lapack support is enabled (`LAPACK_ENABLE` is ON).

`CMAKE_Fortran_FLAGS` - Flags for Fortran compiler
Default:

`CMAKE_Fortran_FLAGS_DEBUG` - Flags used by the compiler during debug builds
Default:

`CMAKE_Fortran_FLAGS_MINSIZEREL` - Flags used by the compiler during release minsize builds
Default:

`CMAKE_Fortran_FLAGS_RELEASE` - Flags used by the compiler during release builds
Default:

`CMAKE_INSTALL_PREFIX` - Install path prefix, prepended onto install directories
Default: /usr/local
Note: The user must have write access to the location specified through this option. Exported SUNDIALS header files and libraries will be installed under subdirectories `include` and `lib` of `CMAKE_INSTALL_PREFIX`, respectively.

`EXAMPLES_ENABLE` - Build the SUNDIALS examples
Default: ON

`EXAMPLES_INSTALL` - Install example files
Default: ON
Note: This option is triggered only if building example programs is enabled (`EXAMPLES_ENABLE` ON). If the user requires installation of example programs then the sources and sample output files for all SUNDIALS modules that are currently enabled will be exported to the directory specified by `EXAMPLES_INSTALL_PATH`. A CMake configuration script will also be automatically generated and exported to the same directory. Additionally, if the configuration is done under a Unix-like system, makefiles for the compilation of the example programs (using the installed SUNDIALS libraries) will be automatically generated and exported to the directory specified by `EXAMPLES_INSTALL_PATH`.

`EXAMPLES_INSTALL_PATH` - Output directory for installing example files
Default: /usr/local/examples
Note: The actual default value for this option will an `examples` subdirectory created under `CMAKE_INSTALL_PREFIX`.

`FCMIX_ENABLE` - Enable Fortran-C support
Default: OFF

`KLU_ENABLE` - Enable KLU support
Default: OFF

`LAPACK_ENABLE` - Enable Lapack support
Default: OFF
Note: Setting this option to ON will trigger the two additional options see below.

`LAPACK_LIBRARIES` - Lapack (and Blas) libraries
Default: /usr/lib/liblapack.so;/usr/lib/libblas.so
Note: CMake will search for these libraries in your `LD_LIBRARY_PATH` prior to searching default system paths.

`MPI_ENABLE` - Enable MPI support
Default: OFF
Note: Setting this option to ON will trigger several additional options related to MPI.

`MPI_MPICC` - `mpicc` program
Default:

`MPI_RUN_COMMAND` - Specify run command for MPI
Default: mpirun
Note: This can either be set to `mpirun` for OpenMPI or `srun` if jobs are managed by `SLURM` - Simple Linux Utility for Resource Management as exists on LLNL's high performance computing clusters.

`MPI_MPIF77` - `mpif77` program
Default:
Note: This option is triggered only if using MPI compiler scripts (`MPI_USE_MPISCRIPTS` is ON) and Fortran-C support is enabled (`FCMIx_ENABLE` is ON).

`OPENMP_ENABLE` - Enable OpenMP support
Default: OFF
Turn on support for the OpenMP based nvector.

`PTHREAD_ENABLE` - Enable Pthreads support
Default: OFF
Turn on support for the Pthreads based nvector.

`SUNDIALS_PRECISION` - Precision used in SUNDIALS, options are: double, single or extended
Default: double

SUPERLUMT_ENABLE - Enable SUPERLU_MT support
Default: OFF

USE_GENERIC_MATH - Use generic (stdc) math libraries
Default: ON

A.1.3 Configuration examples

The following examples will help demonstrate usage of the CMake configure options.
To configure SUNDIALS using the default C and Fortran compilers, and default `mpicc` and `mpif77` parallel compilers, enable compilation of examples, and install libraries, headers, and example sources under subdirectories of `/home/myname/sundials/`, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> /home/myname/sundials/srcdir
%
% make install
%
```

To disable installation of the examples, use:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DMPI_ENABLE=ON \
> -DFCMIX_ENABLE=ON \
> -DEXAMPLES_INSTALL=OFF \
> /home/myname/sundials/srcdir
%
% make install
%
```

A.1.4 Working with external Libraries

The SUNDIALS Suite contains many options to enable implementation flexibility when developing solutions. The following are some notes addressing specific configurations when using the supported third party libraries.

Building with LAPACK and BLAS

To enable LAPACK and BLAS libraries, set the `LAPACK_ENABLE` option to `ON`. If the directory containing the LAPACK and BLAS libraries is in the `LD_LIBRARY_PATH` environment variable, CMake will set the `LAPACK_LIBRARIES` variable accordingly, otherwise CMake will attemp to find the LAPACK and BLAS libraries in standard system locations. To explicitly tell CMake what libraries to use, the `LAPACK_LIBRARIES` varible can be set to the desired libraries. Example:

```
% cmake \
> -DCMAKE_INSTALL_PREFIX=/home/myname/sundials/instdir \
> -DEXAMPLES_INSTALL_PATH=/home/myname/sundials/instdir/examples \
> -DLAPACK_LIBRARIES=/mypath/lib/liblapack.so;/mypath/lib/libblas.so \
> /home/myname/sundials/srcdir
%
% make install
%
```

Building with KLU

The KLU libraries are part of SuiteSparse, a suite of sparse matrix software, available from the Texas A&M University website: `http://faculty.cse.tamu.edu/davis/suitesparse.html`. SUNDIALS has been tested with SuiteSparse version 4.2.1. To enable KLU, set `KLU_ENABLE` to `ON`, set `KLU_INCLUDE_DIR` to the `include` path of the KLU installation and set `KLU_LIBRARY_DIR` to the `lib` path of the KLU installation. The CMake configure will result in populating the following variables: `AMD_LIBRARY`, `AMD_LIBRARY_DIR`, `BTF_LIBRARY`, `BTF_LIBRARY_DIR`, `COLAMD_LIBRARY`, `COLAMD_LIBRARY_DIR`, and `KLU_LIBRARY`

Building with SuperLU_MT

The SuperLU_MT libraries are available for download from the Lawrence Berkeley National Laboratory website: `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/#superlu_mt`. SUNDIALS has been tested with SuperLU_MT version 2.4. To enable SuperLU_MT, set `SUPERLUMT_ENABLE` to `ON`, set `SUPERLUMT_INCLUDE_DIR` to the `SRC` path of the SuperLU_MT installation, and set the variable `SUPERLUMT_LIBRARY_DIR` to the `lib` path of the SuperLU_MT installation. At the same time, the variable `SUPERLUMT_THREAD_TYPE` must be set to either `Pthread` or `OpenMP`.
Do not mix thread types when building SUNDIALS solvers. If threading is enabled for SUNDIALS by having either `OPENMP_ENABLE` or `PTHREAD_ENABLE` set to `ON` then SuperLU_MT should be set to use the same threading type.

A.2 Building and Running Examples

Each of the SUNDIALS solvers is distributed with a set of examples demonstrating basic usage. To build and install the examples, set both `EXAMPLES_ENABLE` and `EXAMPLES_INSTALL` to `ON`. Specify the installation path for the examples with the variable `EXAMPLES_INSTALL_PATH`. CMake will generate `CMakeLists.txt` configuration files (and `Makefile` files if on Linux/Unix) that reference the *installed* SUNDIALS headers and libraries.

Either the `CMakeLists.txt` file or the traditional `Makefile` may be used to build the examples as well as serve as a template for creating user developed solutions. To use the supplied `Makefile` simply run `make` to compile and generate the executables. To use CMake from within the installed example directory, run `cmake` (or `ccmake` to use the GUI) followed by `make` to compile the example code. Note that if CMake is used, it will overwrite the traditional `Makefile` with a new CMake-generated `Makefile`. The resulting output from running the examples can be compared with example output bundled in the SUNDIALS distribution.
NOTE: There will potentially be differences in the output due to machine architecture, compiler versions, use of third party libraries etc.

A.3 Configuring, building, and installing on Windows

CMake can also be used to build SUNDIALS on Windows. To build SUNDIALS for use with Visual Studio the following steps should be performed:

1. Unzip the downloaded tar file(s) into a directory. This will be the *srcdir*
2. Create a separate *builddir*
3. Open a Visual Studio Command Prompt and cd to *builddir*
4. Run cmake-gui ../*srcdir*
   (a) Hit Configure
   (b) Check/Uncheck solvers to be built
   (c) Change CMAKE_INSTALL_PREFIX to *instdir*

   (d) Set other options as desired
   (e) Hit Generate

5. Back in the VS Command Window:
   (a) Run msbuild ALL_BUILD.vcxproj
   (b) Run msbuild INSTALL.vcxproj

The resulting libraries will be in the *instdir*. The SUNDIALS project can also now be opened in Visual Studio. Double click on the ALL_BUILD.vcxproj file to open the project. Build the whole *solution* to create the SUNDIALS libraries. To use the SUNDIALS libraries in your own projects, you must set the include directories for your project, add the SUNDIALS libraries to your project solution, and set the SUNDIALS libraries as dependencies for your project.

A.4 Installed libraries and exported header files

Using the CMake SUNDIALS build system, the command

```
% make install
```

will install the libraries under *libdir* and the public header files under *includedir*. The values for these directories are *instdir*/`lib` and *instdir*/`include`, respectively. The location can be changed by setting the CMake variable `CMAKE_INSTALL_PREFIX`. Although all installed libraries reside under *libdir*/`lib`, the public header files are further organized into subdirectories under *includedir*/`include`.

The installed libraries and exported header files are listed for reference in Tables A.1 and A.2. The file extension *.lib* is typically `.so` for shared libraries and `.a` for static libraries. Note that, in the Tables, names are relative to *libdir* for libraries and to *includedir* for header files.

A typical user program need not explicitly include any of the shared SUNDIALS header files from under the *includedir*/`include/sundials` directory since they are explicitly included by the appropriate solver header files (*e.g.*, `cvode_dense.h` includes `sundials_dense.h`). However, it is both legal and safe to do so, and would be useful, for example, if the functions declared in `sundials_dense.h` are to be used in building a preconditioner.

Table A.1: SUNDIALS libraries and header files

SHARED	Libraries	n/a	
	Header files	sundials/sundials_config.h	sundials/sundials_types.h
		sundials/sundials_math.h	
		sundials/sundials_nvector.h	sundials/sundials_fnvector.h
		sundials/sundials_direct.h	sundials/sundials_lapack.h
		sundials/sundials_dense.h	sundials/sundials_band.h
		sundials/sundials_sparse.h	
		sundials/sundials_iterative.h	sundials/sundials_spgmr.h
		sundials/sundials_spbcgs.h	sundials/sundials_sptfqmr.h
		sundials/sundials_pcg.h	sundials/sundials_spfgmr.h
NVECTOR_SERIAL	Libraries	libsundials_nvecserial.*lib*	libsundials_fnvecserial.a
	Header files	nvector/nvector_serial.h	
NVECTOR_PARALLEL	Libraries	libsundials_nvecparallel.*lib*	libsundials_fnvecparallel.a
	Header files	nvector/nvector_parallel.h	
NVECTOR_OPENMP	Libraries	libsundials_nvecopenmp.*lib*	libsundials_fnvecopenmp.a
	Header files	nvector/nvector_openmp.h	
NVECTOR_PTHREADS	Libraries	libsundials_nvecpthreads.*lib*	libsundials_fnvecpthreads.a
	Header files	nvector/nvector_pthreads.h	
CVODE	Libraries	libsundials_cvode.*lib*	libsundials_fcvode.a
	Header files	cvode/cvode.h	cvode/cvode_impl.h
		cvode/cvode_direct.h	cvode/cvode_lapack.h
		cvode/cvode_dense.h	cvode/cvode_band.h
		cvode/cvode_diag.h	
		cvode/cvode_sparse.h	cvode/cvode_klu.h
		cvode/cvode_superlumt.h	
		cvode/cvode_spils.h	cvode/cvode_spgmr.h
		cvode/cvode_sptfqmr.h	cvode/cvode_spbcgs.h
		cvode/cvode_bandpre.h	cvode/cvode_bbdpre.h
CVODES	Libraries	libsundials_cvodes.*lib*	
	Header files	cvodes/cvodes.h	cvodes/cvodes_impl.h
		cvodes/cvodes_direct.h	cvodes/cvodes_lapack.h
		cvodes/cvodes_dense.h	cvodes/cvodes_band.h
		cvodes/cvodes_diag.h	
		cvodes/cvodes_sparse.h	cvodes/cvodes_klu.h
		cvodes/cvodes_superlumt.h	
		cvodes/cvodes_spils.h	cvodes/cvodes_spgmr.h
		cvodes/cvodes_sptfqmr.h	cvodes/cvodes_spbcgs.h
		cvodes/cvodes_bandpre.h	cvodes/cvodes_bbdpre.h
ARKODE	Libraries	libsundials_arkode.*lib*	libsundials_farkode.a
	Header files	arkode/arkode.h	arkode/arkode_impl.h
		arkode/arkode_direct.h	arkode/arkode_lapack.h
		arkode/arkode_dense.h	arkode/arkode_band.h
		arkode/arkode_sparse.h	arkode/arkode_klu.h
		arkode/arkode_superlumt.h	
		arkode/arkode_spils.h	arkode/arkode_spgmr.h
		arkode/arkode_sptfqmr.h	arkode/arkode_spbcgs.h
		arkode/arkode_pcg.h	arkode/arkode_spfgmr.h
		arkode/arkode_bandpre.h	arkode/arkode_bbdpre.h

Table A.2: SUNDIALS libraries and header files (cont.)

IDA	Libraries	libsundials_ida.*lib*	libsundials_fida.a
	Header files	ida/ida.h	ida/ida_impl.h
		ida/ida_direct.h	ida/ida_lapack.h
		ida/ida_dense.h	ida/ida_band.h
		ida/ida_sparse.h	ida/ida_klu.h
		ida/ida_superlumt.h	
		ida/ida_spils.h	ida/ida_spgmr.h
		ida/ida_spbcgs.h	ida/ida_sptfqmr.h
		ida/ida_bbdpre.h	
IDAS	Libraries	libsundials_idas.*lib*	
	Header files	idas/idas.h	idas/idas_impl.h
		idas/idas_direct.h	idas/idas_lapack.h
		idas/idas_dense.h	idas/idas_band.h
		idas/idas_sparse.h	idas/idas_klu.h
		idas/idas_superlumt.h	
		idas/idas_spils.h	idas/idas_spgmr.h
		idas/idas_spbcgs.h	idas/idas_sptfqmr.h
		idas/idas_bbdpre.h	
KINSOL	Libraries	libsundials_kinsol.*lib*	libsundials_fkinsol.a
	Header files	kinsol/kinsol.h	kinsol/kinsol_impl.h
		kinsol/kinsol_direct.h	kinsol/kinsol_lapack.h
		kinsol/kinsol_dense.h	kinsol/kinsol_band.h
		kinsol/kinsol_sparse.h	kinsol/kinsol_klu.h
		kinsol/kinsol_superlumt.h	
		kinsol/kinsol_spils.h	kinsol/kinsol_spgmr.h
		kinsol/kinsol_spbcgs.h	kinsol/kinsol_sptfqmr.h
		kinsol/kinsol_bbdpre.h	kinsol/kinsol_spfgmr.h

Appendix B

KINSOL Constants

Below we list all input and output constants used by the main solver and linear solver modules, together with their numerical values and a short description of their meaning.

B.1 KINSOL input constants

	KINSOL main solver module	
`KIN_ETACHOICE1`	1	Use Eisenstat and Walker Choice 1 for η.
`KIN_ETACHOICE2`	2	Use Eisenstat and Walker Choice 2 for η.
`KIN_ETACONSTANT`	3	Use constant value for η.
`KIN_NONE`	0	Use inexact Newton globalization.
`KIN_LINESEARCH`	1	Use linesearch globalization.
	Iterative linear solver module	
`PREC_NONE`	0	No preconditioning
`PREC_RIGHT`	2	Preconditioning on the right.
`MODIFIED_GS`	1	Use modified Gram-Schmidt procedure.
`CLASSICAL_GS`	2	Use classical Gram-Schmidt procedure.

B.2 KINSOL output constants

	KINSOL main solver module	
`KIN_SUCCESS`	0	Successful function return.
`KIN_INITIAL_GUESS_OK`	1	The initial user-supplied guess already satisfies the stopping criterion.
`KIN_STEP_LT_STPTOL`	2	The stopping tolerance on scaled step length was satisfied.
`KIN_WARNING`	99	A non-fatal warning. The solver will continue.
`KIN_MEM_NULL`	-1	The `kin_mem` argument was `NULL`.
`KIN_ILL_INPUT`	-2	One of the function inputs is illegal.
`KIN_NO_MALLOC`	-3	The KINSOL memory was not allocated by a call to `KINMalloc`.
`KIN_MEM_FAIL`	-4	A memory allocation failed.
`KIN_LINESEARCH_NONCONV`	-5	The linesearch algorithm was unable to find an iterate sufficiently distinct from the current iterate.

`KIN_MAXITER_REACHED`	-6	The maximum number of nonlinear iterations has been reached.
`KIN_MXNEWT_5X_EXCEEDED`	-7	Five consecutive steps have been taken that satisfy a scaled step length test.
`KIN_LINESEARCH_BCFAIL`	-8	The linesearch algorithm was unable to satisfy the β-condition for `nbcfails` iterations.
`KIN_LINSOLV_NO_RECOVERY`	-9	The user-supplied routine preconditioner slve function failed recoverably, but the preconditioner is already current.
`KIN_LINIT_FAIL`	-10	The linear solver's initialization function failed.
`KIN_LSETUP_FAIL`	-11	The linear solver's setup function failed in an unrecoverable manner.
`KIN_LSOLVE_FAIL`	-12	The linear solver's solve function failed in an unrecoverable manner.
`KIN_SYSFUNC_FAIL`	-13	The system function failed in an unrecoverable manner.
`KIN_FIRST_SYSFUNC_ERR`	-14	The system function failed recoverably at the first call.
`KIN_REPTD_SYSFUNC_ERR`	-15	The system function had repeated recoverable errors.

KINDLS **linear solver module**

`KINDLS_SUCCESS`	0	Successful function return.
`KINDLS_MEM_NULL`	-1	The `kin_mem` argument was `NULL`.
`KINDLS_LMEM_NULL`	-2	The KINDLS linear solver has not been initialized.
`KINDLS_ILL_INPUT`	-3	The KINDLS solver is not compatible with the current NVECTOR module.
`KINDLS_MEM_FAIL`	-4	A memory allocation request failed.
`KINDLS_JACFUNC_UNRECVR`	-5	The Jacobian function failed in an unrecoverable manner.
`KINDLS_JACFUNC_RECVR`	-6	The Jacobian function had a recoverable error.

KINSLS **linear solver module**

`KINSLS_SUCCESS`	0	Successful function return.
`KINSLS_MEM_NULL`	-1	The `kin_mem` argument was `NULL`.
`KINSLS_LMEM_NULL`	-2	The KINSLS linear solver has not been initialized.
`KINSLS_ILL_INPUT`	-3	The KINSLS solver is not compatible with the current NVECTOR module or other input is invalid.
`KINSLS_MEM_FAIL`	-4	A memory allocation request failed.
`KINSLS_JAC_NOSET`	-5	The Jacobian evaluation routine was not been set before the linear solver setup routine was called.
`KINSLS_PACKAGE_FAIL`	-6	An external package call return a failure error code.
`KINSLS_JACFUNC_UNRECVR`	-7	The Jacobian function failed in an unrecoverable manner.
`KINSLS_JACFUNC_RECVR`	-8	The Jacobian function had a recoverable error.

KINSPILS **linear solver modules**

`KINSPILS_SUCCESS`	0	Successful function return.
`KINSPILS_MEM_NULL`	-1	The `kin_mem` argument was `NULL`.
`KINSPILS_LMEM_NULL`	-2	The KINSPILS linear solver has not been initialized.
`KINSPILS_ILL_INPUT`	-3	The KINSPILS solver is not compatible with the current NVECTOR module, or an input value was illegal.

KINSPILS_MEM_FAIL	-4	A memory allocation request failed.
KINSPILS_PMEM_NULL	-5	The preconditioner module has not been initialized.
		SPGMR generic linear solver module
SPGMR_SUCCESS	0	Converged.
SPGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPGMR_CONV_FAIL	2	Failure to converge.
SPGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPGMR_PSET_FAIL_REC	6	The preconditioner setup routine failed recoverably.
SPGMR_MEM_NULL	-1	The SPGMR memory is NULL
SPGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPGMR_PSET_FAIL_UNREC	-6	The preconditioner setup routine failed unrecoverably.
		SPFGMR generic linear solver module (only available in KINSOL and ARKODE)
SPFGMR_SUCCESS	0	Converged.
SPFGMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPFGMR_CONV_FAIL	2	Failure to converge.
SPFGMR_QRFACT_FAIL	3	A singular matrix was found during the QR factorization.
SPFGMR_PSOLVE_FAIL_REC	4	The preconditioner solve function failed recoverably.
SPFGMR_ATIMES_FAIL_REC	5	The Jacobian-times-vector function failed recoverably.
SPFGMR_PSET_FAIL_REC	6	The preconditioner setup routine failed recoverably.
SPFGMR_MEM_NULL	-1	The SPFGMR memory is NULL
SPFGMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPFGMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPFGMR_GS_FAIL	-4	Failure in the Gram-Schmidt procedure.
SPFGMR_QRSOL_FAIL	-5	The matrix R was found to be singular during the QR solve phase.
SPFGMR_PSET_FAIL_UNREC	-6	The preconditioner setup routine failed unrecoverably.
		SPBCG generic linear solver module
SPBCG_SUCCESS	0	Converged.
SPBCG_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPBCG_CONV_FAIL	2	Failure to converge.
SPBCG_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPBCG_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPBCG_PSET_FAIL_REC	5	The preconditioner setup routine failed recoverably.
SPBCG_MEM_NULL	-1	The SPBCG memory is NULL
SPBCG_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed unrecoverably.
SPBCG_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.

SPBCG_PSET_FAIL_UNREC	-4	The preconditioner setup routine failed unrecoverably.

SPTFQMR generic linear solver module

SPTFQMR_SUCCESS	0	Converged.
SPTFQMR_RES_REDUCED	1	No convergence, but the residual norm was reduced.
SPTFQMR_CONV_FAIL	2	Failure to converge.
SPTFQMR_PSOLVE_FAIL_REC	3	The preconditioner solve function failed recoverably.
SPTFQMR_ATIMES_FAIL_REC	4	The Jacobian-times-vector function failed recoverably.
SPTFQMR_PSET_FAIL_REC	5	The preconditioner setup routine failed recoverably.
SPTFQMR_MEM_NULL	-1	The SPTFQMR memory is NULL
SPTFQMR_ATIMES_FAIL_UNREC	-2	The Jacobian-times-vector function failed.
SPTFQMR_PSOLVE_FAIL_UNREC	-3	The preconditioner solve function failed unrecoverably.
SPTFQMR_PSET_FAIL_UNREC	-4	The preconditioner setup routine failed unrecoverably.

Bibliography

[1] KLU Sparse Matrix Factorization Library. http://faculty.cse.tamu.edu/davis/suitesparse.html.

[2] SuperLU_MT Threaded Sparse Matrix Factorization Library. http://crd-legacy.lbl.gov/ xiaoye/-SuperLU/.

[3] D. G. Anderson. Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery*, 12:547–560, 1965.

[4] P. N. Brown. A local convergence theory for combined inexact-Newton/finite difference projection methods. *SIAM J. Numer. Anal.*, 24(2):407–434, 1987.

[5] P. N. Brown and Y. Saad. Hybrid Krylov Methods for Nonlinear Systems of Equations. *SIAM J. Sci. Stat. Comput.*, 11:450–481, 1990.

[6] A. M. Collier and R. Serban. Example Programs for KINSOL v2.7.0. Technical Report UCRL-SM-208114, LLNL, 2011.

[7] T. A. Davis and P. N. Ekanathan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3), 2010.

[8] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton Methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

[9] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[10] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM, Philadelphia, 1996.

[11] S. C. Eisenstat and H. F. Walker. Choosing the Forcing Terms in an Inexact Newton Method. *SIAM J. Sci. Comput.*, 17:16–32, 1996.

[12] H. Fang and Y. Saad. Two classes of secant methods for nonlinear acceleration. *Numer. Linear Algebra Appl.*, 16:197–221, 2009.

[13] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM J. Sci. Comp.*, 14:470–482, 1993.

[14] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS, suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, (31):363–396, 2005.

[15] C. T. Kelley. *Iterative Methods for Solving Linear and Nonlinear Equations*. SIAM, Philadelphia, 1995.

[16] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[17] P. A. Lott, H. F. Walker, C. S. Woodward, and U. M. Yang. An accelerated Picard method for nonlinear systems related to variably saturated flow. *Adv. Wat. Resour.*, 38:92–101, 2012.

[18] J. M. Ortega and W. C. Rheinbolt. *Iterative solution of nonlinear equations in several variables.* SIAM, Philadelphia, 2000. Originally published in 1970 by Academic Press.

[19] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.*, 14(2):461–469, 1993.

[20] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7:856–869, 1986.

[21] H. A. Van Der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13:631–644, 1992.

[22] H. F. Walker and P. Ni. Anderson acceleration for fixed-point iterations. *SIAM Jour. Num. Anal.*, 49(4):1715–1735, 2011.

Index